ESMValTool User's and Developer's Guide

Release 2.11.0.dev0+g11e5b1e.d20231219

ESMValTool Development Team

Dec 19, 2023

ESMVALTOOL

I Getting started 1

1 Installation 3

2 Configuration files 7

3 Input data 21

4 Running 33

5 Output 37

II Example notebooks 41

6 Composing recipes 45

7 Discovering data 55

8 Loading, processing, and visualizing data 59

III The recipe format 63

9 Overview 65

10 Preprocessor 77

IV Diagnostic script interfaces 115

11 Provenance 119

12 Information provided by ESMValCore to the diagnostic script 121

13 Information provided by the diagnostic script to ESMValCore 123

V Development 125

14 Preprocessor function 129

15 Fixing data 135

16 Deriving a variable 143

VI Contributions are very welcome 145

17 Getting started 149

18 Checklist for pull requests 151

19 Scientific relevance 153

20 Pull request title and label 155

21 Code quality 157

22 Documentation 159

23 Tests 161

24 Backward compatibility 163

25 Dependencies 165

26 List of authors 167

27 Pull request checks 169

28 Making a release 171

VII ESMValCore API Reference 175

29 CMOR functions 179

30 Configuration 199

31 Dataset 205

32 Find and download files from ESGF 211

33 Exceptions 217

34 Iris helper functions 219

35 Find files on the local filesystem 221

36 Preprocessor functions 223

37 Type hints 257

38 Experimental API 259

VIII Changelog 277

39 v2.10.0 279

40 v2.9.0 283

41 v2.8.1 287

42 v2.8.0 289

43 v2.7.1 295

44 v2.7.0 297

45 v2.6.0 299

46 v2.5.0 303

47 v2.4.0 307

48 v2.3.1 311

49 v2.3.0 313

50 v2.2.0 317

51 v2.1.0 321

52 v2.0.0 323

53 v2.0.0b9 327

IX Indices and tables 329

Python Module Index 333

Index 335

Part I

Getting started

CHAPTER

ONE

INSTALLATION

1.1 Conda installation

In order to install the Conda package, you will need to install Conda first. For a minimal conda installation (recommended) go to https://conda.io/miniconda.html. It is recommended that you always use the latest version of conda, as problems have been reported when trying to use older versions.

Once you have installed conda, you can install ESMValCore by running:

```
conda install -c conda-forge esmvalcore
```

It is also possible to create a new Conda environment and install ESMValCore into it with a single command:

```
conda create --name esmvalcore -c conda-forge esmvalcore 'python=3.10'
```

Don't forget to activate the newly created environment after the installation:

```
conda activate esmvalcore
```

Of course it is also possible to choose a different name than `esmvalcore` for the environment.

Note: Creating a new Conda environment is often much faster and more reliable than trying to update an existing Conda environment.

1.2 Pip installation

It is also possible to install ESMValCore from PyPI. However, this requires first installing dependencies that are not available on PyPI in some other way. By far the easiest way to install these dependencies is to use conda. For a minimal conda installation (recommended) go to https://conda.io/miniconda.html.

After installing Conda, download the file with the list of dependencies:

```
wget https://raw.githubusercontent.com/ESMValGroup/ESMValCore/main/environment.yml
```

and install these dependencies into a new conda environment with the command

```
conda env create --name esmvalcore --file environment.yml
```

Finally, activate the newly created environment

```
conda activate esmvalcore
```

and install ESMValCore as well as any remaining dependencies with the command:

```
pip install esmvalcore
```

1.3 Docker installation

ESMValCore is also provided through DockerHub in the form of docker containers. See https://docs.docker.com for more information about docker containers and how to run them.

You can get the latest release with

```
docker pull esmvalgroup/esmvalcore:stable
```

If you want to use the current main branch, use

```
docker pull esmvalgroup/esmvalcore:latest
```

To run a container using those images, use:

```
docker run esmvalgroup/esmvalcore:stable --help
```

Note that the container does not see the data or environmental variables available in the host by default. You can make data available with `-v /path:/path/in/container` and environmental variables with `-e VARNAME`.

For example, the following command would run a recipe

```
docker run -e HOME -v "$HOME":"$HOME" -v /data:/data esmvalgroup/esmvalcore:stable -c ~/
↪config-user.yml ~/recipes/recipe_example.yml
```

with the environmental variable `$HOME` available inside the container and the data in the directories `$HOME` and `/data`, so these can be used to find the configuration file, recipe, and data.

It might be useful to define a bash alias or script to abbreviate the above command, for example

```
alias esmvaltool="docker run -e HOME -v $HOME:$HOME -v /data:/data esmvalgroup/
↪esmvalcore:stable"
```

would allow using the `esmvaltool` command without even noticing that the tool is running inside a Docker container.

1.4 Singularity installation

Docker is usually forbidden in clusters due to security reasons. However, there is a more secure alternative to run containers that is usually available on them: Singularity.

Singularity can use docker containers directly from DockerHub with the following command

```
singularity run docker://esmvalgroup/esmvalcore:stable -c ~/config-user.yml ~/recipes/
↪recipe_example.yml
```

Note that the container does not see the data available in the host by default. You can make host data available with `-B /path:/path/in/container`.

It might be useful to define a bash alias or script to abbreviate the above command, for example

```
alias esmvaltool="singularity run -B $HOME:$HOME -B /data:/data docker://esmvalgroup/
↪esmvalcore:stable"
```

would allow using the `esmvaltool` command without even noticing that the tool is running inside a Singularity container.

Some clusters may not allow to connect to external services, in those cases you can first create a singularity image locally:

```
singularity build esmvalcore.sif docker://esmvalgroup/esmvalcore:stable
```

and then upload the image file `esmvalcore.sif` to the cluster. To run the container using the image file `esmvalcore.sif` use:

```
singularity run esmvalcore.sif -c ~/config-user.yml ~/recipes/recipe_example.yml
```

1.5 Installation from source

Note: If you would like to install the development version of ESMValCore alongside ESMValTool, please have a look at these instructions.

To install from source for development, follow these instructions.

- Download and install conda (this should be done even if the system in use already has a preinstalled version of conda, as problems have been reported with using older versions of conda)
- To make the `conda` command available, add `source <prefix>/etc/profile.d/conda.sh` to your `.bashrc` file and restart your shell. If using (t)csh shell, add `source <prefix>/etc/profile.d/conda.csh` to your `.cshrc/.tcshrc` file instead.
- Update conda: `conda update -y conda`
- Clone the ESMValCore Git repository: `git clone https://github.com/ESMValGroup/ESMValCore.git`
- Go to the source code directory: `cd ESMValCore`
- Create the esmvalcore conda environment `conda env create --name esmvalcore --file environment.yml`
- Activate the esmvalcore environment: `conda activate esmvalcore`
- Install in development mode: `pip install -e '.[develop]'`. If you are installing behind a proxy that does not trust the usual pip-urls you can declare them with the option `--trusted-host`, e.g. `pip install --trusted-host=pypi.python.org --trusted-host=pypi.org --trusted-host=files.pythonhosted.org -e .[develop]`
- Test that your installation was successful by running `esmvaltool -h`.

1.6 Pre-installed versions on HPC clusters / other servers

If you would like to use pre-installed versions on HPC clusters (currently CEDA-JASMIN and DKRZ-Levante), and other servers (currently Met Office Linux estate), please have a look at these instructions.

1.7 Installation from the conda lock file

A fast conda environment creation is possible using the provided conda lock files. This is a secure alternative to the installation from source, whenever the conda environment can not be created for some reason. A conda lock file is an explicit environment file that contains pointers to dependency packages as they are hosted on the Anaconda cloud; these have frozen version numbers, build hashes, and channel names, parameters established at the time of the conda lock file creation, so may be obsolete after a while, but they allow for a robust environment creation while they're still up-to-date. We regenerate these lock files every 10 days through automatic Pull Requests (or more frequently, since the automatic generator runs on merges on the main branch too), so to minimize the risk of dependencies becoming obsolete. Conda environment creation from a lock file is done just like with any other environment file:

```
conda create --name esmvaltool --file conda-linux-64.lock
```

The latest, most up-to-date file can always be downloaded directly from the source code repository, a direct download link can be found here.

Note: *pip* and *conda* are NOT installed, so you will have to install them in the new environment: use conda-forge as channel): `conda install -c conda-forge pip` at the very minimum so we can install *esmvalcore* afterwards.

1.8 Creating a conda lock file

We provide a conda lock file for Linux-based operating systems, but if you prefer to build a conda lock file yourself, install the *conda-lock* package first:

```
conda install -c conda-forge conda-lock
```

then run

```
conda-lock lock --platform linux-64 -f environment.yml --mamba
```

(mamba activated for speed) to create a conda lock file for Linux platforms, or run

```
conda-lock lock --platform osx-64 -f environment.yml --mamba
```

to create a lock file for OSX platforms. Note however, that using conda lock files on OSX is still problematic!

CHAPTER

TWO

CONFIGURATION FILES

2.1 Overview

There are several configuration files in ESMValCore:

- `config-user.yml`: sets a number of user-specific options like desired graphical output format, root paths to data, etc.;
- `config-developer.yml`: sets a number of standardized file-naming and paths to data formatting;

and one configuration file which is distributed with ESMValTool:

- `config-references.yml`: stores information on diagnostic and recipe authors and scientific journals references;

2.2 User configuration file

The `config-user.yml` configuration file contains all the global level information needed by ESMValTool. It can be reused as many times the user needs to before changing any of the options stored in it. This file is essentially the gateway between the user and the machine-specific instructions to `esmvaltool`. By default, esmvaltool looks for it in the home directory, inside the `.esmvaltool` folder.

Users can get a copy of this file with default values by running

```
esmvaltool config get-config-user --path=${TARGET_FOLDER}
```

If the option `--path` is omitted, the file will be created in `${HOME}/.esmvaltool`

The following shows the default settings from the `config-user.yml` file with explanations in a commented line above each option. If only certain values are allowed for an option, these are listed after `---`. The option in square brackets is the default value, i.e., the one that is used if this option is omitted in the file.

```
# Destination directory where all output will be written
# Includes log files and performance stats.
output_dir: ~/esmvaltool_output

# Auxiliary data directory
# Used by some recipes to look for additional datasets.
auxiliary_data_dir: ~/auxiliary_data

# Automatic data download from ESGF --- [never]/when_missing/always
# Use automatic download of missing CMIP3, CMIP5, CMIP6, CORDEX, and obs4MIPs
```

(continues on next page)

(continued from previous page)

```
# data from ESGF. ``never`` disables this feature, which is useful if you are
# working on a computer without an internet connection, or if you have limited
# disk space. ``when_missing`` enables the automatic download for files that
# are not available locally. ``always`` will always check ESGF for the latest
# version of a file, and will only use local files if they correspond to that
# latest version.
search_esgf: never

# Directory for storing downloaded climate data
# Make sure to use a directory where you can store multiple GBs of data. Your
# home directory on a HPC is usually not suited for this purpose, so please
# change the default value in this case!
download_dir: ~/climate_data

# Rootpaths to the data from different projects
# This default setting will work if files have been downloaded by ESMValTool
# via ``search_esgf``. Lists are also possible. For site-specific entries,
# see the default ``config-user.yml`` file that can be installed with the
# command ``esmvaltool config get_config_user``. For each project, this can
# be either a single path or a list of paths. Comment out these when using a
# site-specific path.
rootpath:
  default: ~/climate_data

# Directory structure for input data --- [default]/ESGF/BADC/DKRZ/ETHZ/etc.
# This default setting will work if files have been downloaded by ESMValTool
# via ``search_esgf``. See ``config-developer.yml`` for definitions. Comment
# out/replace as per needed.
drs:
  CMIP3: ESGF
  CMIP5: ESGF
  CMIP6: ESGF
  CORDEX: ESGF
  obs4MIPs: ESGF

# Run at most this many tasks in parallel --- [null]/1/2/3/4/...
# Set to ``null`` to use the number of available CPUs. If you run out of
# memory, try setting max_parallel_tasks to ``1`` and check the amount of
# memory you need for that by inspecting the file ``run/resource_usage.txt`` in
# the output directory. Using the number there you can increase the number of
# parallel tasks again to a reasonable number for the amount of memory
# available in your system.
max_parallel_tasks: null

# Log level of the console --- debug/[info]/warning/error
# For much more information printed to screen set log_level to ``debug``.
log_level: info

# Exit on warning --- true/[false]
# Only used in NCL diagnostic scripts.
exit_on_warning: false
```

(continues on next page)

(continued from previous page)

```
# Plot file format --- [png]/pdf/ps/eps/epsi
output_file_type: png

# Remove the ``preproc`` directory if the run was successful --- [true]/false
# By default this option is set to ``true``, so all preprocessor output files
# will be removed after a successful run. Set to ``false`` if you need those files.
remove_preproc_dir: true

# Use netCDF compression --- true/[false]
compress_netcdf: false

# Save intermediary cubes in the preprocessor --- true/[false]
# Setting this to ``true`` will save the output cube from each preprocessing
# step. These files are numbered according to the preprocessing order.
save_intermediary_cubes: false

# Use a profiling tool for the diagnostic run --- [false]/true
# A profiler tells you which functions in your code take most time to run.
# For this purpose we use ``vprof``, see below for notes. Only available for
# Python diagnostics.
profile_diagnostic: false

# Path to custom ``config-developer.yml`` file
# This can be used to customise project configurations. See
# ``config-developer.yml`` for an example. Set to ``null`` to use the default.
config_developer_file: null
```

The `search_esgf` setting can be used to disable or enable automatic downloads from ESGF. If `search_esgf` is set to `never`, the tool does not download any data from the ESGF. If `search_esgf` is set to `when_missing`, the tool will download any CMIP3, CMIP5, CMIP6, CORDEX, and obs4MIPs data that is required to run a recipe but not available locally and store it in `download_dir` using the `ESGF` directory structure defined in the *Developer configuration file*. If `search_esgf` is set to `always`, the tool will first check the ESGF for the needed data, regardless of any local data availability; if the data found on ESGF is newer than the local data (if any) or the user specifies a version of the data that is available only from the ESGF, then that data will be downloaded; otherwise, local data will be used.

The `auxiliary_data_dir` setting is the path to place any required additional auxiliary data files. This is necessary because certain Python toolkits, such as cartopy, will attempt to download data files at run time, typically geographic data files such as coastlines or land surface maps. This can fail if the machine does not have access to the wider internet. This location allows the user to specify where to find such files if they can not be downloaded at runtime. The example user configuration file already contains two valid locations for `auxiliary_data_dir` directories on CEDA-JASMIN and DKRZ, and a number of such maps and shapefiles (used by current diagnostics) are already there. You will need `esmeval` group workspace membership to access the JASMIN one (see instructions how to gain access to the group workspace.

Warning: This setting is not for model or observational datasets, rather it is for extra data files such as shapefiles or other data sources needed by the diagnostics.

The `profile_diagnostic` setting triggers profiling of Python diagnostics, this will tell you which functions in the diagnostic took most time to run. For this purpose we use vprof. For each diagnostic script in the recipe, the profiler writes a `.json` file that can be used to plot a flame graph of the profiling information by running

```
vprof --input-file esmvaltool_output/recipe_output/run/diagnostic/script/profile.json
```

Note that it is also possible to use vprof to understand other resources used while running the diagnostic, including execution time of different code blocks and memory usage.

A detailed explanation of the data finding-related sections of the `config-user.yml` (`rootpath` and `drs`) is presented in the *Data retrieval* section. This section relates directly to the data finding capabilities of ESMValTool and are very important to be understood by the user.

Note: You can choose your `config-user.yml` file at run time, so you could have several of them available with different purposes. One for a formalised run, another for debugging, etc. You can even provide any config user value as a run flag `--argument_name argument_value`

2.3 Dask distributed configuration

The *preprocessor functions* and many of the Python diagnostics in ESMValTool make use of the Iris library to work with the data. In Iris, data can be either real or lazy. Lazy data is represented by dask arrays. Dask arrays consist of many small numpy arrays (called chunks) and if possible, computations are run on those small arrays in parallel. In order to figure out what needs to be computed when, Dask makes use of a 'scheduler'. The default scheduler in Dask is rather basic, so it can only run on a single computer and it may not always find the optimal task scheduling solution, resulting in excessive memory use when using e.g. the *esmvalcore.preprocessor.multi_model_statistics()* preprocessor function. Therefore it is recommended that you take a moment to configure the Dask distributed scheduler. A Dask scheduler and the 'workers' running the actual computations, are collectively called a 'Dask cluster'.

In ESMValCore, the Dask cluster can configured by creating a file called `~/.esmvaltool/dask.yml`, where `~` is short for your home directory. In this file, under the `client` keyword, the arguments to `distributed.Client` can be provided. Under the `cluster` keyword, the type of cluster (e.g. `distributed.LocalCluster`), as well as any arguments required to start the cluster can be provided. Extensive documentation on setting up Dask Clusters is available here.

Warning: The format of the `~/.esmvaltool/dask.yml` configuration file is not yet fixed and may change in the next release of ESMValCore.

Note: If not all preprocessor functions support lazy data, computational performance may be best with the default scheduler. See issue #674 for progress on making all preprocessor functions lazy.

Example configurations

Personal computer

Create a Dask distributed cluster on the computer running ESMValCore using all available resources:

```
cluster:
  type: distributed.LocalCluster
```

this should work well for most personal computers.

Note: Note that, if running this configuration on a shared node of an HPC cluster, Dask will try and use as many

resources it can find available, and this may lead to overcrowding the node by a single user (you)!

Shared computer

Create a Dask distributed cluster on the computer running ESMValCore, with 2 workers with 4 threads/4 GiB of memory each (8 GiB in total):

```
cluster:
  type: distributed.LocalCluster
  n_workers: 2
  threads_per_worker: 4
  memory_limit: 4 GiB
```

this should work well for shared computers.

Computer cluster

Create a Dask distributed cluster on the Levante supercomputer using the Dask-Jobqueue package:

```
cluster:
  type: dask_jobqueue.SLURMCluster
  queue: shared
  account: bk1088
  cores: 8
  memory: 7680MiB
  processes: 2
  interface: ib0
  local_directory: "/scratch/b/b381141/dask-tmp"
  n_workers: 24
```

This will start 24 workers with `cores / processes = 4` threads each, resulting in `n_workers / processes = 12` Slurm jobs, where each Slurm job will request 8 CPU cores and 7680 MiB of memory and start `processes = 2` workers. This example will use the fast infiniband network connection (called `ib0` on Levante) for communication between workers running on different nodes. It is important to set the right location for temporary storage, in this case the `/scratch` space is used. It is also possible to use environmental variables to configure the temporary storage location, if you cluster provides these.

A configuration like this should work well for larger computations where it is advantageous to use multiple nodes in a compute cluster. See Deploying Dask Clusters on High Performance Computers for more information.

Externally managed Dask cluster

Use an externally managed cluster, e.g. a cluster that you started using the Dask Jupyterlab extension:

```
client:
  address: '127.0.0.1:8786'
```

See here for an example of how to configure this on a remote system.

For debugging purposes, it can be useful to start the cluster outside of ESMValCore because then Dask dashboard remains available after ESMValCore has finished running.

Advice on choosing performant configurations

The threads within a single worker can access the same memory locations, so they may freely pass around chunks, while communicating a chunk between workers is done by copying it, so this is (a bit) slower. Therefore it is beneficial for performance to have multiple threads per worker. However, due to limitations in the CPython implementation (known as the Global Interpreter Lock or GIL), only a single thread in a worker can execute Python code (this limitation does

not apply to compiled code called by Python code, e.g. numpy), therefore the best performing configurations will typically not use much more than 10 threads per worker.

Due to limitations of the NetCDF library (it is not thread-safe), only one of the threads in a worker can read or write to a NetCDF file at a time. Therefore, it may be beneficial to use fewer threads per worker if the computation is very simple and the runtime is determined by the speed with which the data can be read from and/or written to disk.

2.4 ESGF configuration

The `esmvaltool run` command can automatically download the files required to run a recipe from ESGF for the projects CMIP3, CMIP5, CMIP6, CORDEX, and obs4MIPs. The downloaded files will be stored in the `download_dir` specified in the *User configuration file*. To enable automatic downloads from ESGF, set `search_esgf: when_missing` or `search_esgf: always` in the *User configuration file*, or provide the corresponding command line arguments `--search_esgf=when_missing` or `--search_esgf=always` when running the recipe.

Note: When running a recipe that uses many or large datasets on a machine that does not have any data available locally, the amount of data that will be downloaded can be in the range of a few hundred gigabyte to a few terrabyte. See Obtaining input data for advice on getting access to machines with large datasets already available.

A log message will be displayed with the total amount of data that will be downloaded before starting the download. If you see that this is more than you would like to download, stop the tool by pressing the `Ctrl` and `C` keys on your keyboard simultaneously several times, edit the recipe so it contains fewer datasets and try again.

For downloading some files (e.g. those produced by the CORDEX project), you need to log in to be able to download the data.

See the ESGF user guide for instructions on how to create an ESGF OpenID account if you do not have one yet. Note that the OpenID account consists of 3 components instead of the usual two, in addition a username and password you also need the hostname of the provider of the ID; for example esgf-data.dkrz.de. Even though the account is issued by a particular host, the same OpenID account can be used to download data from all hosts in the ESGF.

Next, configure your system so the `esmvaltool` can use your credentials. This can be done using the *keyring* package or they can be stored in a *configuration file*.

2.4.1 Storing credentials in keyring

First install the keyring package. Note that this requires a supported backend that may not be available on compute clusters, see the keyring documentation for more information.

```
pip install keyring
```

Next, set your username and password by running the commands:

```
keyring set ESGF hostname
keyring set ESGF username
keyring set ESGF password
```

for example, if you created an account on the host esgf-data.dkrz.de with username 'cookiemonster' and password 'Welcome01', run the command

```
keyring set ESGF hostname
```

this will display the text

```
Password for 'hostname' in 'ESGF':
```

type `esgf-data.dkrz.de` (the characters will not be shown) and press `Enter`. Repeat the same procedure with `keyring set ESGF username`, type `cookiemonster` and press `Enter` and `keyring set ESGF password`, type `Welcome01` and press `Enter`.

To check that you entered your credentials correctly, run:

```
keyring get ESGF hostname
keyring get ESGF username
keyring get ESGF password
```

2.4.2 Configuration file

An optional configuration file can be created for configuring how the tool uses esgf-pyclient to find and download data. The name of this file is `~/.esmvaltool/esgf-pyclient.yml`.

Logon

In the `logon` section you can provide arguments that will be passed on to `pyesgf.logon.LogonManager.logon()`. For example, you can store the hostname, username, and password or your OpenID account in the file like this:

```
logon:
  hostname: "your-hostname"
  username: "your-username"
  password: "your-password"
```

for example

```
logon:
  hostname: "esgf-data.dkrz.de"
  username: "cookiemonster"
  password: "Welcome01"
```

if you created an account on the host esgf-data.dkrz.de with username 'cookiemonster' and password 'Welcome01'. Alternatively, you can configure an interactive log in:

```
logon:
  interactive: true
```

Note that storing your password in plain text in the configuration file is less secure. On shared systems, make sure the permissions of the file are set so only you and administrators can read it, i.e.

```
ls -l ~/.esmvaltool/esgf-pyclient.yml
```

shows permissions `-rw-------`.

Search

Any arguments to pyesgf.search.connection.SearchConnection can be provided in the section search_connection, for example:

```
search_connection:
  expire_after: 2592000  # the number of seconds in a month
```

to keep cached search results for a month.

The default settings are:

```
urls:
  - 'https://esgf.ceda.ac.uk/esg-search'
  - 'https://esgf-node.llnl.gov/esg-search'
  - 'https://esgf-data.dkrz.de/esg-search'
  - 'https://esgf-node.ipsl.upmc.fr/esg-search'
  - 'https://esg-dn1.nsc.liu.se/esg-search'
  - 'https://esgf.nci.org.au/esg-search'
  - 'https://esgf.nccs.nasa.gov/esg-search'
  - 'https://esgdata.gfdl.noaa.gov/esg-search'
distrib: true
timeout: 120  # seconds
cache: '~/.esmvaltool/cache/pyesgf-search-results'
expire_after: 86400  # cache expires after 1 day
```

Note that by default the tool will try the ESGF index nodes in the order provided in the configuration file and use the first one that is online. Some ESGF index nodes may return search results faster than others, so you may be able to speed up the search for files by experimenting with placing different index nodes at the top of the list.

If you experience errors while searching, it sometimes helps to delete the cached results.

2.4.3 Download statistics

The tool will maintain statistics of how fast data can be downloaded from what host in the file ~/.esmvaltool/cache/esgf-hosts.yml and automatically select hosts that are faster. There is no need to manually edit this file, though it can be useful to delete it if you move your computer to a location that is very different from the place where you previously downloaded data. An entry in the file might look like this:

```
esgf2.dkrz.de:
  duration (s): 8
  error: false
  size (bytes): 69067460
  speed (MB/s): 7.9
```

The tool only uses the duration and size to determine the download speed, the speed shown in the file is not used. If error is set to true, the most recent download request to that host failed and the tool will automatically try this host only as a last resort.

2.5 Developer configuration file

Most users and diagnostic developers will not need to change this file, but it may be useful to understand its content. It will be installed along with ESMValCore and can also be viewed on GitHub: esmvalcore/config-developer.yml. This configuration file describes the file system structure and CMOR tables for several key projects (CMIP6, CMIP5, obs4MIPs, OBS6, OBS) on several key machines (e.g. BADC, CP4CDS, DKRZ, ETHZ, SMHI, BSC), and for native output data for some models (ICON, IPSL, … see *Configuring datasets in native format*). CMIP data is stored as part of the Earth System Grid Federation (ESGF) and the standards for file naming and paths to files are set out by CMOR and DRS. For a detailed description of these standards and their adoption in ESMValCore, we refer the user to *CMIP data* section where we relate these standards to the data retrieval mechanism of the ESMValCore.

By default, esmvaltool looks for it in the home directory, inside the '.esmvaltool' folder.

Users can get a copy of this file with default values by running

```
esmvaltool config get-config-developer --path=${TARGET_FOLDER}
```

If the option `--path` is omitted, the file will be created in `${HOME}/.esmvaltool`.

Note: Remember to change your config-user file if you want to use a custom config-developer.

Example of the CMIP6 project configuration:

```
CMIP6:
  input_dir:
    default: '/'
    BADC: '{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/
↪{version}'
    DKRZ: '{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/
↪{version}'
    ETHZ: '{exp}/{mip}/{short_name}/{dataset}/{ensemble}/{grid}/'
  input_file: '{short_name}_{mip}_{dataset}_{exp}_{ensemble}_{grid}*.nc'
  output_file: '{project}_{dataset}_{mip}_{exp}_{ensemble}_{short_name}'
  cmor_type: 'CMIP6'
  cmor_strict: true
```

2.5.1 Input file paths

When looking for input files, the `esmvaltool` command provided by ESMValCore replaces the placeholders `{item}` in `input_dir` and `input_file` with the values supplied in the recipe. ESMValCore will try to automatically fill in the values for institute, frequency, and modeling_realm based on the information provided in the CMOR tables and/or *extra_facets* when reading the recipe. If this fails for some reason, these values can be provided in the recipe too.

The data directory structure of the CMIP projects is set up differently at each site. As an example, the CMIP6 directory path on BADC would be:

```
'{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/{version}'
```

The resulting directory path would look something like this:

```
CMIP/MOHC/HadGEM3-GC31-LL/historical/r1i1p1f3/Omon/tos/gn/latest
```

Please, bear in mind that `input_dirs` can also be a list for those cases in which may be needed:

```
- '{exp}/{ensemble}/original/{mip}/{short_name}/{grid}/{version}'
- '{exp}/{ensemble}/computed/{mip}/{short_name}/{grid}/{version}'
```

In that case, the resultant directories will be:

```
historical/r1i1p1f3/original/Omon/tos/gn/latest
historical/r1i1p1f3/computed/Omon/tos/gn/latest
```

For a more in-depth description of how to configure ESMValCore so it can find your data please see *CMIP data*.

2.5.2 Preprocessor output files

The filename to use for preprocessed data is configured in a similar manner using `output_file`. Note that the extension `.nc` (and if applicable, a start and end time) will automatically be appended to the filename.

2.5.3 Project CMOR table configuration

ESMValCore comes bundled with several CMOR tables, which are stored in the directory esmvalcore/cmor/tables. These are copies of the tables available from PCMDI.

For every `project` that can be used in the recipe, there are four settings related to CMOR table settings available:

- `cmor_type`: can be `CMIP5` if the CMOR table is in the same format as the CMIP5 table or `CMIP6` if the table is in the same format as the CMIP6 table.
- `cmor_strict`: if this is set to `false`, the CMOR table will be extended with variables from the *Custom CMOR tables* (by default loaded from the `esmvalcore/cmor/tables/custom` directory) and it is possible to use variables with a `mip` which is different from the MIP table in which they are defined.
- `cmor_path`: path to the CMOR table. Relative paths are with respect to esmvalcore/cmor/tables. Defaults to the value provided in `cmor_type` written in lower case.
- `cmor_default_table_prefix`: Prefix that needs to be added to the `mip` to get the name of the file containing the `mip` table. Defaults to the value provided in `cmor_type`.

2.5.4 Custom CMOR tables

As mentioned in the previous section, the CMOR tables of projects that use `cmor_strict: false` will be extended with custom CMOR tables. By default, these are loaded from esmvalcore/cmor/tables/custom. However, by using the special project `custom` in the `config-developer.yml` file with the option `cmor_path`, a custom location for these custom CMOR tables can be specified:

```
custom:
  cmor_path: ~/my/own/custom_tables
```

This path can be given as relative path (relative to esmvalcore/cmor/tables) or as absolute path. Other options given for this special table will be ignored.

Custom tables in this directory need to follow the naming convention `CMOR_{short_name}.dat` and need to be given in CMIP5 format.

Example for the file `CMOR_asr.dat`:

```
SOURCE: CMIP5
!=============
variable_entry:    asr
!=============
modeling_realm:    atmos
!---------------------------------
! Variable attributes:
!---------------------------------
standard_name:
units:             W m-2
cell_methods:      time: mean
cell_measures:     area: areacella
long_name:         Absorbed shortwave radiation
!---------------------------------
! Additional variable information:
!---------------------------------
dimensions:        longitude latitude time
type:              real
positive:          down
!---------------------------------
!
```

It is also possible to use a special coordinates file `CMOR_coordinates.dat`. If this is not present in the custom directory, the one from the default directory (esmvalcore/cmor/tables/custom/CMOR_coordinates.dat) is used.

2.5.5 Filter preprocessor warnings

It is possible to ignore specific warnings of the preprocessor for a given `project`. This is particularly useful for native datasets which do not follow the CMOR standard by default and consequently produce a lot of warnings when handled by Iris. This can be configured in the `config-developer.yml` file for some steps of the preprocessing chain.

Currently supported preprocessor steps:

- *load()*

Here is an example on how to ignore specific warnings during the preprocessor step `load` for all datasets of project `EMAC` (taken from the default `config-developer.yml` file):

```
ignore_warnings:
  load:
    - {message: 'Missing CF-netCDF formula term variable .*, referenced by netCDF␣
↪variable .*', module: iris}
    - {message: 'Ignored formula of unrecognised type: .*', module: iris}
```

The keyword arguments specified in the list items are directly passed to `warnings.filterwarnings()` in addition to `action=ignore` (may be overwritten in `config-developer.yml`).

2.5.6 Configuring datasets in native format

ESMValCore can be configured for handling native model output formats and specific reanalysis/observation datasets without preliminary reformatting. These datasets can be either hosted under the `native6` project (mostly native reanalysis/observational datasets) or under a dedicated project, e.g., `ICON` (mostly native models).

Example:

```
native6:
  cmor_strict: false
  input_dir:
    default: 'Tier{tier}/{dataset}/{version}/{frequency}/{short_name}'
  input_file:
    default: '*.nc'
  output_file: '{project}_{dataset}_{type}_{version}_{mip}_{short_name}'
  cmor_type: 'CMIP6'
  cmor_default_table_prefix: 'CMIP6_'

ICON:
  cmor_strict: false
  input_dir:
    default:
      - '{exp}'
      - '{exp}/outdata'
  input_file:
    default: '{exp}_{var_type}*.nc'
  output_file: '{project}_{dataset}_{exp}_{var_type}_{mip}_{short_name}'
  cmor_type: 'CMIP6'
  cmor_default_table_prefix: 'CMIP6_'
```

A detailed description on how to add support for further native datasets is given *here*.

Hint: When using native datasets, it might be helpful to specify a custom location for the *Custom CMOR tables*. This allows reading arbitrary variables from native datasets. Note that this requires the option `cmor_strict: false` in the *project configuration* used for the native model output.

2.6 References configuration file

The esmvaltool/config-references.yml file contains the list of ESMValTool diagnostic and recipe authors, references and projects. Each author, project and reference referred to in the documentation section of a recipe needs to be in this file in the relevant section.

For instance, the recipe `recipe_ocean_example.yml` file contains the following documentation section:

```
documentation:
  authors:
    - demo_le

  maintainer:
    - demo_le
```

(continues on next page)

(continued from previous page)

```
  references:
    - demora2018gmd

  projects:
    - ukesm
```

These four items here are named people, references and projects listed in the `config-references.yml` file.

2.7 Extra Facets

It can be useful to automatically add extra key-value pairs to variables or datasets in the recipe. These key-value pairs can be used for *finding data* or for providing extra information to the functions that *fix data* before passing it on to the preprocessor.

To support this, we provide the extra facets facilities. Facets are the key-value pairs described in *Recipe section: datasets*. Extra facets allows for the addition of more details per project, dataset, mip table, and variable name.

More precisely, one can provide this information in an extra yaml file, named *{project}-something.yml*, where *{project}* corresponds to the project as used by ESMValTool in *Recipe section: datasets* and "something" is arbitrary.

2.7.1 Format of the extra facets files

The extra facets are given in a yaml file, whose file name identifies the project. Inside the file there is a hierarchy of nested dictionaries with the following levels. At the top there is the *dataset* facet, followed by the *mip* table, and finally the *short_name*. The leaf dictionary placed here gives the extra facets that will be made available to data finder and the fix infrastructure. The following example illustrates the concept.

Listing 1: Extra facet example file *native6-era5.yml*

```
ERA5:
  Amon:
    tas: {source_var_name: "t2m", cds_var_name: "2m_temperature"}
```

The three levels of keys in this mapping can contain Unix shell-style wildcards. The special characters used in shell-style wildcards are:

Pattern	Meaning
*	matches everything
?	matches any single character
[seq]	matches any character in `seq`
[!seq]	matches any character not in `seq`

where `seq` can either be a sequence of characters or just a bunch of characters, for example `[A-C]` matches the characters `A`, `B`, and `C`, while `[AC]` matches the characters `A` and `C`.

For example, this is used to automatically add `product: output1` to any variable of any CMIP5 dataset that does not have a `product` key yet:

Listing 2: Extra facet example file cmip5-product.yml

```
'*':
  '*':
    '*': {product: output1}
```

2.7.2 Location of the extra facets files

Extra facets files can be placed in several different places. When we use them to support a particular use-case within the ESMValTool project, they will be provided in the sub-folder *extra_facets* inside the package `esmvalcore.config`. If they are used from the user side, they can be either placed in *~/.esmvaltool/extra_facets* or in any other directory of the users choosing. In that case this directory must be added to the *config-user.yml* file under the *extra_facets_dir* setting, which can take a single directory or a list of directories.

The order in which the directories are searched is

1. The internal directory *esmvalcore.config/extra_facets*
2. The default user directory *~/.esmvaltool/extra_facets*
3. The custom user directories in the order in which they are given in *config-user.yml*.

The extra facets files within each of these directories are processed in lexicographical order according to their file name.

In all cases it is allowed to supersede information from earlier files in later files. This makes it possible for the user to effectively override even internal default facets, for example to deal with local particularities in the data handling.

2.7.3 Use of extra facets

For extra facets to be useful, the information that they provide must be applied. There are fundamentally two places where this comes into play. One is *the datafinder*, the other are *fixes*.

CHAPTER

THREE

INPUT DATA

3.1 Overview

Data discovery and retrieval is the first step in any evaluation process; ESMValTool uses a *semi-automated* data finding mechanism with inputs from both the user configuration file and the recipe file: this means that the user will have to provide the tool with a set of parameters related to the data needed and once these parameters have been provided, the tool will automatically find the right data. We will detail below the data finding and retrieval process and the input the user needs to specify, giving examples on how to use the data finding routine under different scenarios.

3.2 Data types

3.2.1 CMIP data

CMIP data is widely available via the Earth System Grid Federation (ESGF) and is accessible to users either via automatic download by `esmvaltool` or through the ESGF data nodes hosted by large computing facilities (like CEDA-Jasmin, DKRZ, etc). This data adheres to, among other standards, the DRS and Controlled Vocabulary standard for naming files and structured paths; the DRS ensures that files and paths to them are named according to a standardized convention. Examples of this convention, also used by ESMValTool for file discovery and data retrieval, include:

- CMIP6 file: `{variable_short_name}_{mip}_{dataset_name}_{experiment}_{ensemble}_{grid}_{start-date}-{e nc`
- CMIP5 file: `{variable_short_name}_{mip}_{dataset_name}_{experiment}_{ensemble}_{start-date}-{end-date nc`
- OBS file: `{project}_{dataset_name}_{type}_{version}_{mip}_{short_name}_{start-date}-{end-date}. nc`

Similar standards exist for the standard paths (input directories); for the ESGF data nodes, these paths differ slightly, for example:

- CMIP6 path for BADC: `ROOT-BADC/{institute}/{dataset_name}/{experiment}/{ensemble}/ {mip}/ {variable_short_name}/{grid}`;
- CMIP6 path for ETHZ: `ROOT-ETHZ/{experiment}/{mip}/{variable_short_name}/{dataset_name}/ {ensemble}/{grid}`

From the ESMValTool user perspective the number of data input parameters is optimized to allow for ease of use. We detail this procedure in the next section.

3.2.2 Observational data

Part of observational data is retrieved in the same manner as CMIP data, for example using the `OBS` root path set to:

```
OBS: /gws/nopw/j04/esmeval/obsdata-v2
```

and the dataset:

```
- {dataset: ERA-Interim, project: OBS6, type: reanaly, version: 1, start_year:␣
↪2014, end_year: 2015, tier: 3}
```

in `recipe.yml` in `datasets` or `additional_datasets`, the rules set in *CMOR-DRS* are used again and the file will be automatically found:

```
/gws/nopw/j04/esmeval/obsdata-v2/Tier3/ERA-Interim/OBS_ERA-Interim_reanaly_1_Amon_ta_
↪201401-201412.nc
```

Since observational data are organized in Tiers depending on their level of public availability, the `default` directory must be structured accordingly with sub-directories `TierX` (`Tier1`, `Tier2` or `Tier3`), even when `drs: default`.

3.2.3 Datasets in native format

Some datasets are supported in their native format (i.e., the data is not formatted according to a CMIP data request) through the `native6` project (mostly native reanalysis/observational datasets) or through a dedicated project, e.g., `ICON` (mostly native models). A detailed description of how to include new native datasets is given *here*.

Hint: When using native datasets, it might be helpful to specify a custom location for the *Custom CMOR tables*. This allows reading arbitrary variables from native datasets. Note that this requires the option `cmor_strict: false` in the *project configuration* used for the native model output.

Supported native reanalysis/observational datasets

The following native reanalysis/observational datasets are supported under the `native6` project. To use these datasets, put the files containing the data in the directory that you have configured for the `native6` project in your *User configuration file*, in a subdirectory called `Tier{tier}/{dataset}/{version}/{frequency}/{short_name}`. Replace the items in curly braces by the values used in the variable/dataset definition in the *recipe*. Below is a list of native reanalysis/observational datasets currently supported.

ERA5

- Supported variables: `cl`, `clt`, `evspsbl`, `evspsblpot`, `mrro`, `pr`, `prsn`, `ps`, `psl`, `ptype`, `rls`, `rlds`, `rsds`, `rsdt`, `rss`, `uas`, `vas`, `tas`, `tasmax`, `tasmin`, `tdps`, `ts`, `tsn` (`E1hr`/`Amon`), `orog` (`fx`)
- Tier: 3

Note: According to the description of Evapotranspiration and potential Evapotranspiration on the Copernicus page (https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-era5-single-levels-monthly-means?tab=overview): "The ECMWF Integrated Forecasting System (IFS) convention is that downward fluxes are positive. Therefore, negative values indicate evaporation and positive values indicate condensation."

In the CMOR table, these fluxes are defined as positive, if they go from the surface into the atmosphere: "Evaporation at surface (also known as evapotranspiration): flux of water into the atmosphere due to conversion of both liquid and solid phases to vapor (from underlying surface and vegetation)." Therefore, the ERA5 (and ERA5-Land) CMORizer switches the signs of `evspsbl` and `evspsblpot` to be compatible with the CMOR standard used e.g. by the CMIP models.

MSWEP

- Supported variables: `pr`
- Supported frequencies: `mon`, `day`, `3hr`.
- Tier: 3

For example for monthly data, place the files in the `/Tier3/MSWEP/version/mon/pr` subdirectory of your `native6` project location.

Note: For monthly data (`V220`), the data must be postfixed with the date, i.e. rename `global_monthly_050deg.nc` to `global_monthly_050deg_197901-201710.nc`

For more info: http://www.gloh2o.org/

Data for the version `V220` can be downloaded from: https://hydrology.princeton.edu/data/hylkeb/MSWEP_V220/.

Supported native models

The following models are natively supported by ESMValCore. In contrast to the native observational datasets listed above, they use dedicated projects instead of the project `native6`.

CESM

ESMValTool is able to read native CESM model output.

Warning: The support for native CESM output is still experimental. Currently, only one variable (*tas*) is fully supported. Other 2D variables might be supported by specifying appropriate facets in the recipe or extra facets files (see text below). 3D variables (data that uses a vertical dimension) are not supported, yet.

The default naming conventions for input directories and files for CESM are

- **input directories: 3 different types supported:**
  - `/` (run directory)
  - `{case}/{gcomp}/hist` (short-term archiving)
  - `{case}/{gcomp}/proc/{tdir}/{tperiod}` (post-processed data)
- input files: `{case}.{scomp}.{type}.{string}*nc`

as configured in the *config-developer file* (using the default DRS `drs: default` in the *User configuration file*). More information about CESM naming conventions are given here.

Note: The `{string}` entry in the input file names above does not only correspond to the (optional) `$string` entry for CESM model output files, but can also be used to read post-processed files. In the latter case, `{string}` corresponds to the combination `$SSTRING.$TSTRING`.

Thus, example dataset entries could look like this:

```
datasets:
  - {project: CESM, dataset: CESM2, case: f.e21.FHIST_BGC.f09_f09_mg17.CMIP6-AMIP.001,␣
↪type: h0, mip: Amon, short_name: tas, start_year: 2000, end_year: 2014}
  - {project: CESM, dataset: CESM2, case: f.e21.F1850_BGC.f09_f09_mg17.CFMIP-hadsst-
↪piForcing.001, type: h0, gcomp: atm, scomp: cam, mip: Amon, short_name: tas, start_
↪year: 2000, end_year: 2014}
```

Variable-specific defaults for the facet `gcomp` and `scomp` are given in the extra facets (see next paragraph) for some variables, but this can be overwritten in the recipe.

Similar to any other fix, the CESM fix allows the use of *extra facets*. By default, the file `cesm-mappings.yml` is used for that purpose. Currently, this file only contains default facets for a single variable (*tas*); for other variables, these entries need to be defined in the recipe. Supported keys for extra facets are:

Key	Description	Default value if not specified
gcomp	Generic component-model name	No default (needs to be specified in extra facets or recipe if default DRS is used)
raw_r	Variable name of the variable in the raw input file	CMOR variable name of the corresponding variable
raw_u	Units of the variable in the raw input file	If specified, the value given by the `units` attribute in the raw input file; otherwise `unknown`
scomp	Specific component-model name	No default (needs to be specified in extra facets or recipe if default DRS is used)
strir	Short string which is used to further identify the history file type (corresponds to `$string` or `$SSTRING.$TSTRING` in the CESM file name conventions; see note above)	`''` (empty string)
tdir	Entry to distinguish time averages from time series from diagnostic plot sets (only used for post-processed data)	`''` (empty string)
tperi	Time period over which the data was processed (only used for post-processed data)	`''` (empty string)

EMAC

ESMValTool is able to read native EMAC model output.

The default naming conventions for input directories and files for EMAC are

- input directories: `{exp}/{channel}`
- input files: `{exp}*{channel}{postproc_flag}.nc`

as configured in the *config-developer file* (using the default DRS `drs: default` in the *User configuration file*).

Thus, example dataset entries could look like this:

```
datasets:
  - {project: EMAC, dataset: EMAC, exp: historical, mip: Amon, short_name: tas, start_
↪year: 2000, end_year: 2014}
  - {project: EMAC, dataset: EMAC, exp: historical, mip: Omon, short_name: tos, postproc_
↪flag: "-p-mm", start_year: 2000, end_year: 2014}
  - {project: EMAC, dataset: EMAC, exp: historical, mip: Amon, short_name: ta, raw_name:␣
↪tm1_p39_cav, start_year: 2000, end_year: 2014}
```

Please note the duplication of the name `EMAC` in `project` and `dataset`, which is necessary to comply with ESMValTool's data finding and CMORizing functionalities. A variable-specific default for the facet `channel` is given in the extra facets (see next paragraph) for many variables, but this can be overwritten in the recipe.

Similar to any other fix, the EMAC fix allows the use of *extra facets*. By default, the file `emac-mappings.yml` is used for that purpose. For some variables, extra facets are necessary; otherwise ESMValTool cannot read them properly. Supported keys for extra facets are:

Key	Description	Default value if not specified
`channel`	Channel in which the desired variable is stored	No default (needs to be specified in extra facets or recipe if default DRS is used)
`postproc_f`	Postprocessing flag of the data	`''` (empty string)
`raw_name`	Variable name of the variable in the raw input file	CMOR variable name of the corresponding variable
`raw_units`	Units of the variable in the raw input file	If specified, the value given by the `units` attribute in the raw input file; otherwise `unknown`

Note: `raw_name` can be given as `str` or `list`. The latter is used to support multiple different variables names in the input file. In this case, the prioritization is given by the order of the list; if possible, use the first entry, if this is not present, use the second, etc. This is particularly useful for files in which regular averages (`*_ave`) or conditional averages (`*_cav`) exist.

For 3D variables defined on pressure levels, only the pressure levels defined by the CMOR table (e.g., for *Amon*'s *ta*: `tm1_p19_cav` and `tm1_p19_ave`) are given in the default extra facets file. If other pressure levels are desired, e.g., `tm1_p39_cav`, this has to be explicitly specified in the recipe using `raw_name: tm1_p39_cav` or `raw_name: [tm1_p19_cav, tm1_p39_cav]`.

ICON

ESMValTool is able to read native ICON model output.

The default naming conventions for input directories and files for ICON are

- input directories: `{exp}` or `{exp}/outdata`
- input files: `{exp}_{var_type}*.nc`

as configured in the *config-developer file* (using the default DRS `drs: default` in the *User configuration file*).

Thus, example dataset entries could look like this:

```
datasets:
  - {project: ICON, dataset: ICON, exp: icon-2.6.1_atm_amip_R2B5_r1i1p1f1,
     mip: Amon, short_name: tas, start_year: 2000, end_year: 2014}
```

(continues on next page)

(continued from previous page)

```
  - {project: ICON, dataset: ICON, exp: historical, mip: Amon,
     short_name: ta, var_type: atm_dyn_3d_ml, start_year: 2000,
     end_year: 2014}
```

Please note the duplication of the name `ICON` in `project` and `dataset`, which is necessary to comply with ESMValTool's data finding and CMORizing functionalities. A variable-specific default for the facet `var_type` is given in the extra facets (see below) for many variables, but this can be overwritten in the recipe. This is necessary if your ICON output is structured in one variable per file. For example, if your output is stored in files called `<exp>_<variable_name>_atm_2d_ml_YYYYMMDDThhmmss.nc`, use `var_type: <variable_name>_atm_2d_ml` in the recipe for this variable.

Usually, ICON reports aggregated values at the end of the corresponding time output intervals. For example, for monthly output, ICON reports the month February as "1 March". Thus, by default, ESMValCore shifts all time points back by 1/2 of the output time interval so that the new time point corresponds to the center of the interval. This can be disabled by using `shift_time: false` in the recipe or the extra facets (see below). For point measurements (identified by `cell_methods = "time: point"`), this is always disabled.

Warning: To get all desired time points, do **not** use `start_year` and `end_year` in the recipe, but rather `timerange` with at least 8 digits. For example, to get data for the years 2000 and 2001, use `timerange: 20000101/20020101` instead of `timerange: 2000/2001` or `start_year: 2000`, `end_year: 2001`. See *Time ranges* for more information on the `timerange` option.

Usually, ESMValCore will need the corresponding ICON grid file of your simulation to work properly (examples: setting latitude/longitude coordinates if these are not yet present, UGRIDization [see below], etc.). This grid file can either be specified as absolute or relative (to `auxiliary_data_dir` as defined in the *User configuration file*) path with the facet `horizontal_grid` in the recipe or the extra facets (see below), or retrieved automatically from the *grid_file_uri* attribute of the input files. In the latter case, the file is downloaded once and then cached. The cached file is valid for 7 days.

ESMValCore can automatically make native ICON data UGRID-compliant when loading the data. The UGRID conventions provide a standardized format to store data on unstructured grids, which is required by many software packages or tools to work correctly. An example is the horizontal regridding of native ICON data to a regular grid. While the built-in *unstructured_nearest scheme* can handle unstructured grids not in UGRID format, using more complex regridding algorithms (for example provided by the iris-esmf-regrid package through *Generic regridding schemes*) requires the input data in UGRID format. The following code snippet provides a preprocessor that regrids native ICON data to a 1°x1° grid using ESMF's first-order conservative regridding algorithm:

```
preprocessors:
  regrid_icon:
    regrid:
      target_grid: 1x1
      scheme:
        reference: esmf_regrid.schemes:ESMFAreaWeighted
```

This automatic UGRIDization is enabled by default, but can be switched off with the facet `ugrid: false` in the recipe or the extra facets (see below). This is useful for diagnostics that do not support input data in UGRID format (yet) like the Psyplot diagnostic or if you want to use the built-in *unstructured_nearest scheme* regridding scheme.

For 3D ICON variables, ESMValCore tries to add the pressure level information (from the variables *pfull* and *phalf*) and/or altitude information (from the variables *zg* and *zghalf*) to the preprocessed output files. If neither of these variables are available in the input files, it is possible to specify the location of files that include the corresponding *zg* or *zghalf* variables with the facets `zg_file` and/or `zghalf_file` in the recipe or the extra facets. The paths to these files can be specified absolute or relative (to `auxiliary_data_dir` as defined in the *User configuration file*).

Hint: To use the *extract_levels()* preprocessor on native ICON data, you need to specify the name of the vertical coordinate (e.g., `coordinate: air_pressure`) since native ICON output usually provides a 3D air pressure field instead of a simple 1D vertical coordinate. This also works if your files only contain altitude information (in this case, the US standard atmosphere is used to convert between altitude and pressure levels; see *Vertical interpolation* for details). Example:

```
preprocessors:
  extract_500hPa_level_from_icon:
    extract_levels:
      levels: 50000
      scheme: linear
      coordinate: air_pressure
```

Similar to any other fix, the ICON fix allows the use of *extra facets*. By default, the file `icon-mappings.yml` is used for that purpose. For some variables, extra facets are necessary; otherwise ESMValTool cannot read them properly. Supported keys for extra facets are:

Key	Description	Default value if not specified
`horizont`	Absolute or relative (to `auxiliary_data_dir` defined in the *User configuration file*) path to the ICON grid file	If not given, use file attribute `grid_file_uri` to retrieve ICON grid file
`latitude`	Standard name of the latitude coordinate in the raw input file	`latitude`
`longitud`	Standard name of the longitude coordinate in the raw input file	`longitude`
`raw_name`	Variable name of the variable in the raw input file	CMOR variable name of the corresponding variable
`raw_unit`	Units of the variable in the raw input file	If specified, the value given by the `units` attribute in the raw input file; otherwise `unknown`
`shift_ti`	Shift time points back by 1/2 of the corresponding output time interval	`True`
`ugrid`	Automatic UGRIDization of the input data	`True`
`var_type`	Variable type of the variable in the raw input file	No default (needs to be specified in extra facets or recipe if default DRS is used)
`zg_file`	Absolute or relative (to `auxiliary_data_dir` defined in the *User configuration file*) path to the input file that contains *zg*	If possible, use *zg* variable provided by the raw input file
`zghalf_f`	Absolute or relative (to `auxiliary_data_dir` defined in the *User configuration file*) path to the input file that contains *zghalf*	If possible, use *zghalf* variable provided by the raw input file

Hint: In order to read cell area files (`areacella` and `areacello`), one additional manual step is necessary: Copy the ICON grid file (you can find a download link in the global attribute `grid_file_uri` of your ICON data) to your ICON input directory and change its name in such a way that only the grid file is found when the cell area variables are required. Make sure that this file is not found when other variables are loaded.

For example, you could use a new `var_type`, e.g., `horizontalgrid` for this file. Thus, an ICON grid file located in `2.6.1_atm_amip_R2B5_r1i1p1f1/2.6.1_atm_amip_R2B5_r1i1p1f1_horizontalgrid.nc` can be found using `var_type: horizontalgrid` in the recipe (assuming the default naming conventions listed above). Make sure that

no other variable uses this `var_type`.

If you want to use the *area_statistics()* preprocessor on *regridded* ICON data, make sure to **not** use the cell area files by using the `skip: true` syntax in the recipe as described in *Supplementary variables (ancillary variables and cell measures)*, e.g.,

```
datasets:
  - {project: ICON, dataset: ICON, exp: amip,
     supplementary_variables: [{short_name: areacella, skip: true}]}
```

IPSL-CM6

Both output formats (i.e. the `Output` and the `Analyse / Time series` formats) are supported, and should be configured in recipes as e.g.:

```
datasets:
  - {simulation: CM61-LR-hist-03.1950, exp: piControl, out: Analyse, freq: TS_MO,
     account: p86caub,  status: PROD, dataset: IPSL-CM6, project: IPSLCM,
     root: /thredds/tgcc/store}
  - {simulation: CM61-LR-hist-03.1950, exp: historical, out: Output, freq: MO,
     account: p86caub,  status: PROD, dataset: IPSL-CM6, project: IPSLCM,
     root: /thredds/tgcc/store}
```

The `Output` format is an example of a case where variables are grouped in multi-variable files, which name cannot be computed directly from datasets attributes alone but requires to use an extra_facets file, which principles are explained in *Extra Facets*, and which content is `available here`. These multi-variable files must also undergo some data selection.

3.3 Data retrieval

Data retrieval in ESMValTool has two main aspects from the user's point of view:

- data can be found by the tool, subject to availability on disk or ESGF;
- it is the user's responsibility to set the correct data retrieval parameters;

The first point is self-explanatory: if the user runs the tool on a machine that has access to a data repository or multiple data repositories, then ESMValTool will look for and find the available data requested by the user. If the files are not found locally, the tool can search the ESGF and download the missing files, provided that they are available.

The second point underlines the fact that the user has full control over what type and the amount of data is needed for the analyses. Setting the data retrieval parameters is explained below.

3.3.1 Enabling automatic downloads from the ESGF

To enable automatic downloads from ESGF, set `search_esgf: when_missing` (use local files whenever possible) or `search_esgf: always` (always search ESGF for latest version of files and only use local data if it is the latest version) in the *User configuration file*, or provide the corresponding command line arguments `--search_esgf=when_missing` or `--search_esgf=always` when running the recipe. The files will be stored in the `download_dir` set in the *User configuration file*.

3.3.2 Setting the correct root paths

The first step towards providing ESMValTool the correct set of parameters for data retrieval is setting the root paths to the data. This is done in the user configuration file `config-user.yml`. The two sections where the user will set the paths are `rootpath` and `drs`. `rootpath` contains pointers to `CMIP`, `OBS`, `default` and `RAWOBS` root paths; `drs` sets the type of directory structure the root paths are structured by. It is important to first discuss the `drs` parameter: as we've seen in the previous section, the DRS as a standard is used for both file naming conventions and for directory structures.

3.3.3 Synda

If the synda install command is used to download data, it maintains the directory structure as on ESGF. To find data downloaded by synda, use the `SYNDA` `drs` parameter.

```
drs:
  CMIP6: SYNDA
  CMIP5: SYNDA
```

3.3.4 Explaining `config-user/drs: CMIP5:` or `config-user/drs: CMIP6:`

Whereas ESMValTool will **always** use the CMOR standard for file naming (please refer above), by setting the `drs` parameter the user tells the tool what type of root paths they need the data from, e.g.:

```
drs:
  CMIP6: BADC
```

will tell the tool that the user needs data from a repository structured according to the BADC DRS structure, i.e.:

`ROOT/{institute}/{dataset_name}/{experiment}/{ensemble}/{mip}/{variable_short_name}/{grid}`;

setting the `ROOT` parameter is explained below. This is a strictly-structured repository tree and if there are any sort of irregularities (e.g. there is no `{mip}` directory) the data will not be found! `BADC` can be replaced with `DKRZ` or `ETHZ` depending on the existing `ROOT` directory structure. The snippet

```
drs:
  CMIP6: default
```

is another way to retrieve data from a `ROOT` directory that has no DRS-like structure; `default` indicates that the data lies in a directory that contains all the files without any structure.

Note: When using `CMIP6: default` or `CMIP5: default` it is important to remember that all the needed files must be in the same top-level directory set by `default` (see below how to set `default`).

3.3.5 Explaining `config-user/rootpath:`

`rootpath` identifies the root directory for different data types (`ROOT` as we used it above):

- `CMIP` e.g. `CMIP5` or `CMIP6`: this is the *root* path(s) to where the CMIP files are stored; it can be a single path or a list of paths; it can point to an ESGF node or it can point to a user private repository. Example for a CMIP5 root path pointing to the ESGF node on CEDA-Jasmin (formerly known as BADC):

  ```
  CMIP5: /badc/cmip5/data/cmip5/output1
  ```

  Example for a CMIP6 root path pointing to the ESGF node on CEDA-Jasmin:

  ```
  CMIP6: /badc/cmip6/data/CMIP6/CMIP
  ```

  Example for a mix of CMIP6 root path pointing to the ESGF node on CEDA-Jasmin and a user-specific data repository for extra data:

  ```
  CMIP6: [/badc/cmip6/data/CMIP6/CMIP, /home/users/johndoe/cmip_data]
  ```

- `OBS`: this is the *root* path(s) to where the observational datasets are stored; again, this could be a single path or a list of paths, just like for CMIP data. Example for the OBS path for a large cache of observation datasets on CEDA-Jasmin:

  ```
  OBS: /gws/nopw/j04/esmeval/obsdata-v2
  ```

- `default`: this is the *root* path(s) where the tool will look for data from projects that do not have their own rootpath set.
- `RAWOBS`: this is the *root* path(s) to where the raw observational data files are stored; this is used by `esmvaltool data format`.

3.3.6 Dataset definitions in `recipe`

Once the correct paths have been established, ESMValTool collects the information on the specific datasets that are needed for the analysis. This information, together with the CMOR convention for naming files (see *CMOR-DRS*) will allow the tool to search and find the right files. The specific datasets are listed in any recipe, under either the `datasets` and/or `additional_datasets` sections, e.g.

```
datasets:
  - {dataset: HadGEM2-CC, project: CMIP5, exp: historical, ensemble: r1i1p1, start_year:␣
↪2001, end_year: 2004}
  - {dataset: UKESM1-0-LL, project: CMIP6, exp: historical, ensemble: r1i1p1f2, grid: gn,
↪ start_year: 2004, end_year: 2014}
```

The data finding feature will use this information to find data for **all** the variables specified in `diagnostics/variables`.

3.4 Recap and example

Let us look at a practical example for a recap of the information above: suppose you are using a `config-user.yml` that has the following entries for data finding:

```
rootpath:  # running on CEDA-Jasmin
  CMIP6: /badc/cmip6/data/CMIP6/CMIP
drs:
  CMIP6: BADC  # since you are on CEDA-Jasmin
```

and the dataset you need is specified in your `recipe.yml` as:

```
- {dataset: UKESM1-0-LL, project: CMIP6, mip: Amon, exp: historical, grid: gn, ensemble:␣
↪r1i1p1f2, start_year: 2004, end_year: 2014}
```

for a variable, e.g.:

```
diagnostics:
  some_diagnostic:
    description: some_description
    variables:
      ta:
        preprocessor: some_preprocessor
```

The tool will then use the root path `/badc/cmip6/data/CMIP6/CMIP` and the dataset information and will assemble the full DRS path using information from *CMOR-DRS* and establish the path to the files as:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon
```

then look for variable `ta` and specifically the latest version of the data file:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon/ta/gn/latest/
```

and finally, using the file naming definition from *CMOR-DRS* find the file:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon/ta/gn/latest/ta_
↪Amon_UKESM1-0-LL_historical_r1i1p1f2_gn_195001-201412.nc
```

3.5 Data loading

Data loading is done using the data load functionality of *iris*; we will not go into too much detail about this since we can point the user to the specific functionality here but we will underline that the initial loading is done by adhering to the CF Conventions that *iris* operates by as well (see CF Conventions Document and the search page for CF standard names).

3.6 Data concatenation from multiple sources

Oftentimes data retrieving results in assembling a continuous data stream from multiple files or even, multiple experiments. The internal mechanism through which the assembly is done is via cube concatenation. One peculiarity of iris concatenation (see iris cube concatenation) is that it doesn't allow for concatenating time-overlapping cubes; this case is rather frequent with data from models overlapping in time, and is accounted for by a function that performs a flexible concatenation between two cubes, depending on the particular setup:

- cubes overlap in time: resulting cube is made up of the overlapping data plus left and right hand sides on each side of the overlapping data; note that in the case of the cubes coming from different experiments the resulting concatenated cube will have composite data made up from multiple experiments: assume [cube1: exp1, cube2: exp2] and cube1 starts before cube2, and cube2 finishes after cube1, then the concatenated cube will be made up of cube2: exp2 plus the section of cube1: exp1 that contains data not provided in cube2: exp2;
- cubes don't overlap in time: data from the two cubes is bolted together;

Note that two cube concatenation is the base operation of an iterative process of reducing multiple cubes from multiple data segments via cube concatenation ie if there is no time-overlapping data, the cubes concatenation is performed in one step.

3.7 Use of extra facets in the datafinder

Extra facets are a mechanism to provide additional information for certain kinds of data. The general approach is described in *Extra Facets*. Here, we describe how they can be used to locate data files within the datafinder framework. This is useful to build paths for directory structures and file names that require more information than what is provided in the recipe. A common application is the location of variables in multi-variable files as often found in climate models' native output formats.

Another use case is files that use different names for variables in their file name than for the netCDF4 variable name.

To apply the extra facets for this purpose, simply use the corresponding tag in the applicable DRS inside the *config-developer.yml* file. For example, given the extra facets in *Extra facet example file native6-era5.yml*, one might write the following.

Listing 1: Example drs use in *config-developer.yml*

```
native6:
  input_file:
    default: '{name_in_filename}*.nc'
```

The same replacement mechanism can be employed everywhere where tags can be used, particularly in *input_dir* and *input_file*.

CHAPTER

FOUR

RUNNING

The ESMValCore package provides the `esmvaltool` command line tool, which can be used to run a *recipe*.

To list the available commands, run

```
esmvaltool --help
```

It is also possible to get help on specific commands, e.g.

```
esmvaltool run --help
```

will display the help message with all options for the `run` command.

To run a recipe, call `esmvaltool run` with the path to the desired recipe:

```
esmvaltool run recipe_example.yml
```

The `esmvaltool run recipe_example.yml` command will first look if `recipe_example.yml` is the path to an existing file. If this is the case, it will run that recipe. If you have ESMValTool installed, it will look if the name matches one of the recipes in your ESMValTool installation directory, in the subdirectory recipes and run that.

Note: There is no `recipe_example.yml` shipped with either ESMValCore or ESMValTool. If you would like to try out the command above, replace `recipe_example.yml` with the path to an existing recipe, e.g. examples/recipe_python.yml if you have the ESMValTool package installed.

To work with installed recipes, the ESMValTool package provides the `esmvaltool recipes` command, see Available diagnostics and metrics.

If the configuration file is not in the default location `~/.esmvaltool/config-user.yml`, you can pass its path explicitly:

```
esmvaltool run --config_file /path/to/config-user.yml recipe_example.yml
```

It is also possible to explicitly change values from the config file using flags:

```
esmvaltool run --argument_name argument_value recipe_example.yml
```

To automatically download the files required to run a recipe from ESGF, set `search_esgf` to `when_missing` (use local files whenever possible) or `always` (always search ESGF for latest version of files and only use local data if it is the latest version) in the *User configuration file* or run the tool with the corresponding commands

```
esmvaltool run --search_esgf=when_missing recipe_example.yml
```

or

```
esmvaltool run --search_esgf=always recipe_example.yml
```

This feature is available for projects that are hosted on the ESGF, i.e. CMIP3, CMIP5, CMIP6, CORDEX, and obs4MIPs.

To control the strictness of the CMOR checker and the checks during concatenation on auxiliary coordinates, supplementary variables, and derived coordinates, use the flag `--check_level`:

```
esmvaltool run --check_level=relaxed recipe_example.yml
```

Possible values are:

- *ignore*: all errors will be reported as warnings. Concatenation will be performed without checks.
- *relaxed*: only fail if there are critical errors. Concatenation will be performed without checks.
- *default*: fail if there are any errors.
- *strict*: fail if there are any warnings.

To re-use pre-processed files from a previous run of the same recipe, you can use

```
esmvaltool run recipe_example.yml --resume_from ~/esmvaltool_output/recipe_python_
↪20210930_123907
```

Multiple directories can be specified for re-use, make sure to quote them:

```
esmvaltool run recipe_example.yml --resume_from "~/esmvaltool_output/recipe_python_
↪20210930_101007 ~/esmvaltool_output/recipe_python_20210930_123907"
```

The first preprocessor directory containing the required data will be used.

This feature can be useful when developing new diagnostics, because it avoids the need to re-run the preprocessor. Another potential use case is running the preprocessing part of a recipe on one or more machines that have access to a lot of data and then running the diagnostics on a machine without access to data.

To run only the preprocessor tasks from a recipe, use

```
esmvaltool run recipe_example.yml --remove_preproc_dir=False --run_diagnostic=False
```

Note: Only preprocessing *tasks* that completed successfully can be re-used with the `--resume_from` option. Preprocessing tasks that completed successfully, contain a file called *metadata.yml* in their output directory.

To run a reduced version of the recipe, usually for testing purpose you can use

```
esmvaltool run --max_datasets=NDATASETS --max_years=NYEARS recipe_example.yml
```

In this case, the recipe will limit the number of datasets per variable to NDATASETS and the total amount of years loaded to NYEARS. They can also be used separately. Note that diagnostics may require specific combinations of available data, so use the above two flags at your own risk and for testing purposes only.

To run a recipe, even if some datasets are not available, use

```
esmvaltool run --skip_nonexistent=True recipe_example.yml
```

It is also possible to select only specific diagnostics to be run. To tun only one, just specify its name. To provide more than one diagnostic to filter use the syntax 'diag1 diag2/script1' or '("diag1", "diag2/script1")' and pay attention to the quotes.

```
esmvaltool run --diagnostics=diagnostic1 recipe_example.yml
```

Note: ESMValTool command line interface is created using the Fire python package. This package supports the creation of completion scripts for the Bash and Fish shells. Go to https://google.github.io/python-fire/using-cli/#python-fires-flags to learn how to set up them.

CHAPTER

FIVE

OUTPUT

ESMValTool automatically generates a new output directory with every run. The location is determined by the output_dir option in the config-user.yml file, the recipe name, and the date and time, using the the format: `YYYYMMDD_HHMMSS`.

For instance, a typical output location would be: `output_directory/recipe_ocean_amoc_20190118_1027/`

This is effectively produced by the combination: `output_dir/recipe_name_YYYYMMDD_HHMMSS/`

This directory will contain 4 further subdirectories:

1. *Diagnostic output* (`work`): A place for any diagnostic script results that are not plots, e.g. files in NetCDF format (depends on the diagnostics).
2. *Plots* (`plots`): The location for all the plots, split by individual diagnostics and fields.
3. *Run* (`run`): This directory includes all log files, a copy of the recipe, a summary of the resource usage, and the *settings.yml* interface files and temporary files created by the diagnostic scripts.
4. *Preprocessed datasets* (`preproc`): This directory contains all the preprocessed netcdfs data and the *metadata.yml* interface files. Note that by default this directory will be deleted after each run, because most users will only need the results from the diagnostic scripts.

A summary of the output is produced in the file: `index.html`

5.1 Preprocessed datasets

The preprocessed datasets will be stored to the preproc/ directory. Each variable in each diagnostic will have its own the *metadata.yml* interface files saved in the preproc directory.

If the option `save_intermediary_cubes` is set to `true` in the config-user.yml file, then the intermediary cubes will also be saved here. This option is set to false in the default `config-user.yml` file.

If the option `remove_preproc_dir` is set to `true` in the config-user.yml file, then the preproc directory will be deleted after the run completes. This option is set to true in the default `config-user.yml` file.

5.2 Run

The log files in the run directory are automatically generated by ESMValTool and create a record of the output messages produced by ESMValTool and they are saved in the run directory. They can be helpful for debugging or monitoring the job, but also allow a record of the job output to screen after the job has been completed.

The run directory will also contain a copy of the recipe and the *settings.yml* file, described below. The run directory is also where the diagnostics are executed, and may also contain several temporary files while diagnostics are running.

5.3 Diagnostic output

The work/ directory will contain all files that are output at the diagnostic stage. Ie, the model data is preprocessed by ESMValTool and stored in the preproc/ directory. These files are opened by the diagnostic script, then some processing is applied. Once the diagnostic level processing has been applied, the results should be saved to the work directory.

5.4 Plots

The plots directory is where diagnostics save their output figures. These plots are saved in the format requested by the option *output_file_type* in the config-user.yml file.

5.5 Settings.yml

The settings.yml file is automatically generated by ESMValTool. Each diagnostic will produce a unique settings.yml file.

The settings.yml file passes several global level keys to diagnostic scripts. This includes several flags from the config-user.yml file (such as 'log_level'), several paths which are specific to the diagnostic being run (such as 'plot_dir' and 'run_dir') and the location on disk of the metadata.yml file (described below).

```
input_files:[[...]recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/
↪metadata.yml]
log_level: debug
output_file_type: png
plot_dir: [...]recipe_ocean_bgc_20190118_134855/plots/diag_timeseries_scalars/Scalar_
↪timeseries
profile_diagnostic: false
recipe: recipe_ocean_bgc.yml
run_dir: [...]recipe_ocean_bgc_20190118_134855/run/diag_timeseries_scalars/Scalar_
↪timeseries
script: Scalar_timeseries
version: 2.0a1
work_dir: [...]recipe_ocean_bgc_20190118_134855/work/diag_timeseries_scalars/Scalar_
↪timeseries
```

The first item in the settings file will be a list of *Metadata.yml* files. There is a metadata.yml file generated for each field in each diagnostic.

5.6 Metadata.yml

The metadata.yml files is automatically generated by ESMValTool. Along with the settings.yml file, it passes all the paths, boolean flags, and additional arguments that your diagnostic needs to know in order to run.

The metadata is loaded from cfg as a dictionairy object in python diagnostics.

Here is an example metadata.yml file:

```
?
  [...]/recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/CMIP5_
↪HadGEM2-ES_Omon_historical_r1i1p1_TO0M_mfo_2002-2004.nc
  : cmor_table: CMIP5
  dataset: HadGEM2-ES
  diagnostic: diag_timeseries_scalars
  end_year: 2004
  ensemble: r1i1p1
  exp: historical
  field: TO0M
  filename: [...]recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/
↪CMIP5_HadGEM2-ES_Omon_historical_r1i1p1_TO0M_mfo_2002-2004.nc
  frequency: mon
  institute: [INPE, MOHC]
  long_name: Sea Water Transport
  mip: Omon
  modeling_realm: [ocean]
  preprocessor: prep_timeseries_scalar
  project: CMIP5
  recipe_dataset_index: 0
  short_name: mfo
  standard_name: sea_water_transport_across_line
  start_year: 2002
  units: kg s-1
  variable_group: mfo
```

As you can see, this is effectively a dictionary with several items including data paths, metadata and other information.

There are several tools available in python which are built to read and parse these files. The tools are available in the shared directory in the diagnostics directory.

Part II

Example notebooks

Example notebooks are available in the notebooks folder and can also be viewed here. These notebooks demonstrate the use of the *Python API*.

CHAPTER

SIX

COMPOSING RECIPES

This notebooks shows how to fill the datasets section in a recipe.

[1]:
```
from esmvalcore.config import CFG
from esmvalcore.dataset import Dataset, datasets_to_recipe
import yaml
```

Configure ESMValCore so it always searches the ESGF for data

[2]:
```
CFG['search_esgf'] = 'always'
```

Here is a small example recipe, that uses the `datasets_to_recipe` function to convert a list of datasets to a recipe:

[3]:
```
tas = Dataset(
    short_name='tas',
    mip='Amon',
    project='CMIP6',
    dataset='CanESM5-1',
    ensemble='r1i1p1f1',
    exp='historical',
    grid='gn',
    timerange='2000/2002',
)
tas['diagnostic'] = 'diagnostic_name'

pr = tas.copy(short_name='pr')

print(yaml.safe_dump(datasets_to_recipe([tas, pr])))
```

```
datasets:
- dataset: CanESM5-1
diagnostics:
  diagnostic_name:
    variables:
      pr:
        ensemble: r1i1p1f1
        exp: historical
        grid: gn
        mip: Amon
        project: CMIP6
        timerange: 2000/2002
      tas:
```

(continues on next page)

(continued from previous page)

```
        ensemble: r1i1p1f1
        exp: historical
        grid: gn
        mip: Amon
        project: CMIP6
        timerange: 2000/2002
```

A more ambitious recipe might want to use all data that is available on ESGF. We can define a dataset template with a facet value of * where any value can be used. This can then be expanded to a list of datasets using the `from_files()` method.

[4]:
```
dataset_template = Dataset(
    short_name='tas',
    mip='Amon',
    project='CMIP6',
    exp='historical',
    dataset='*',
    institute='*',
    ensemble='*',
    grid='*',
)
datasets = list(dataset_template.from_files())
len(datasets)
```

[4]:
```
778
```

This results in the following recipe:

[5]:
```
for dataset in datasets:
    dataset.facets['diagnostic'] = 'diagnostic_name'
print(yaml.safe_dump(datasets_to_recipe(datasets)))
```

```
datasets:
- dataset: TaiESM1
  ensemble: r(1:2)i1p1f1
  grid: gn
  institute: AS-RCEC
- dataset: AWI-CM-1-1-MR
  ensemble: r(1:5)i1p1f1
  grid: gn
  institute: AWI
- dataset: AWI-ESM-1-1-LR
  ensemble: r1i1p1f1
  grid: gn
  institute: AWI
- dataset: BCC-CSM2-MR
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: BCC
- dataset: BCC-ESM1
  ensemble: r(1:3)i1p1f1
  grid: gn
```

(continues on next page)

(continued from previous page)

```
  institute: BCC
- dataset: CAMS-CSM1-0
  ensemble: r1i1p1f2
  grid: gn
  institute: CAMS
- dataset: CAMS-CSM1-0
  ensemble: r(1:2)i1p1f1
  grid: gn
  institute: CAMS
- dataset: CAS-ESM2-0
  ensemble: r(1:4)i1p1f1
  grid: gn
  institute: CAS
- dataset: FGOALS-f3-L
  ensemble: r(1:3)i1p1f1
  grid: gr
  institute: CAS
- dataset: FGOALS-g3
  ensemble: r(1:6)i1p1f1
  grid: gn
  institute: CAS
- dataset: IITM-ESM
  ensemble: r1i1p1f1
  grid: gn
  institute: CCCR-IITM
- dataset: CanESM5-1
  ensemble: r(1:20)i1p1f1
  grid: gn
  institute: CCCma
- dataset: CanESM5-1
  ensemble: r(1:25)i1p2f1
  grid: gn
  institute: CCCma
- dataset: CanESM5-1
  ensemble: r22i1p1f1
  grid: gn
  institute: CCCma
- dataset: CanESM5-1
  ensemble: r(24:39)i1p1f1
  grid: gn
  institute: CCCma
- dataset: CanESM5-1
  ensemble: r(41:50)i1p1f1
  grid: gn
  institute: CCCma
- dataset: CanESM5-CanOE
  ensemble: r(1:3)i1p2f1
  grid: gn
  institute: CCCma
- dataset: CanESM5
  ensemble: r(1:25)i1p1f1
  grid: gn
```

(continues on next page)

(continued from previous page)

```
  institute: CCCma
- dataset: CanESM5
  ensemble: r(1:40)i1p2f1
  grid: gn
  institute: CCCma
- dataset: CMCC-CM2-HR4
  ensemble: r1i1p1f1
  grid: gn
  institute: CMCC
- dataset: CMCC-CM2-SR5
  ensemble: r1i1p1f1
  grid: gn
  institute: CMCC
- dataset: CMCC-CM2-SR5
  ensemble: r(2:11)i1p2f1
  grid: gn
  institute: CMCC
- dataset: CMCC-ESM2
  ensemble: r1i1p1f1
  grid: gn
  institute: CMCC
- dataset: CNRM-CM6-1-HR
  ensemble: r1i1p1f2
  grid: gr
  institute: CNRM-CERFACS
- dataset: CNRM-CM6-1
  ensemble: r(1:30)i1p1f2
  grid: gr
  institute: CNRM-CERFACS
- dataset: CNRM-ESM2-1
  ensemble: r(1:11)i1p1f2
  grid: gr
  institute: CNRM-CERFACS
- dataset: ACCESS-CM2
  ensemble: r(1:10)i1p1f1
  grid: gn
  institute: CSIRO-ARCCSS
- dataset: ACCESS-ESM1-5
  ensemble: r(1:40)i1p1f1
  grid: gn
  institute: CSIRO
- dataset: E3SM-1-0
  ensemble: r(1:5)i1p1f1
  grid: gr
  institute: E3SM-Project
- dataset: E3SM-1-1-ECA
  ensemble: r1i1p1f1
  grid: gr
  institute: E3SM-Project
- dataset: E3SM-1-1
  ensemble: r1i1p1f1
  grid: gr
```

(continues on next page)

(continued from previous page)

```
  institute: E3SM-Project
- dataset: E3SM-2-0
  ensemble: r(1:5)i1p1f1
  grid: gr
  institute: E3SM-Project
- dataset: EC-Earth3-AerChem
  ensemble: r1i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-AerChem
  ensemble: r(3:4)i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-CC
  ensemble: r1i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-CC
  ensemble: r4i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-CC
  ensemble: r(6:13)i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-Veg-LR
  ensemble: r(1:3)i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-Veg
  ensemble: r(1:6)i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-Veg
  ensemble: r10i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-Veg
  ensemble: r12i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3-Veg
  ensemble: r14i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3
  ensemble: r(1:7)i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: EC-Earth3
  ensemble: r(9:25)i1p1f1
  grid: gr
```

(continues on next page)

(continued from previous page)

```
  institute: EC-Earth-Consortium
- dataset: EC-Earth3
  ensemble: r(101:150)i1p1f1
  grid: gr
  institute: EC-Earth-Consortium
- dataset: FIO-ESM-2-0
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: FIO-QLNM
- dataset: MPI-ESM-1-2-HAM
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: HAMMOZ-Consortium
- dataset: INM-CM4-8
  ensemble: r1i1p1f1
  grid: gr1
  institute: INM
- dataset: INM-CM5-0
  ensemble: r(1:10)i1p1f1
  grid: gr1
  institute: INM
- dataset: IPSL-CM5A2-INCA
  ensemble: r1i1p1f1
  grid: gr
  institute: IPSL
- dataset: IPSL-CM6A-LR-INCA
  ensemble: r1i1p1f1
  grid: gr
  institute: IPSL
- dataset: IPSL-CM6A-LR
  ensemble: r(1:33)i1p1f1
  grid: gr
  institute: IPSL
- dataset: KIOST-ESM
  ensemble: r1i1p1f1
  grid: gr1
  institute: KIOST
- dataset: MIROC-ES2H
  ensemble: r1i1p(1:3)f2
  grid: gn
  institute: MIROC
- dataset: MIROC-ES2H
  ensemble: r(1:3)i1p4f2
  grid: gn
  institute: MIROC
- dataset: MIROC-ES2L
  ensemble: r1i1000p1f2
  grid: gn
  institute: MIROC
- dataset: MIROC-ES2L
  ensemble: r(1:30)i1p1f2
  grid: gn
```

(continues on next page)

(continued from previous page)

```
  institute: MIROC
- dataset: MIROC6
  ensemble: r(1:50)i1p1f1
  grid: gn
  institute: MIROC
- dataset: HadGEM3-GC31-LL
  ensemble: r(1:5)i1p1f3
  grid: gn
  institute: MOHC
- dataset: HadGEM3-GC31-MM
  ensemble: r(1:4)i1p1f3
  grid: gn
  institute: MOHC
- dataset: UKESM1-0-LL
  ensemble: r(1:4)i1p1f2
  grid: gn
  institute: MOHC
- dataset: UKESM1-0-LL
  ensemble: r(5:7)i1p1f3
  grid: gn
  institute: MOHC
- dataset: UKESM1-0-LL
  ensemble: r(8:12)i1p1f2
  grid: gn
  institute: MOHC
- dataset: UKESM1-0-LL
  ensemble: r(16:19)i1p1f2
  grid: gn
  institute: MOHC
- dataset: UKESM1-1-LL
  ensemble: r1i1p1f2
  grid: gn
  institute: MOHC
- dataset: ICON-ESM-LR
  ensemble: r(1:5)i1p1f1
  grid: gn
  institute: MPI-M
- dataset: MPI-ESM1-2-HR
  ensemble: r(1:10)i1p1f1
  grid: gn
  institute: MPI-M
- dataset: MPI-ESM1-2-LR
  ensemble: r1i2000p1f1
  grid: gn
  institute: MPI-M
- dataset: MPI-ESM1-2-LR
  ensemble: r(1:30)i1p1f1
  grid: gn
  institute: MPI-M
- dataset: MRI-ESM2-0
  ensemble: r1i2p1f1
  grid: gn
```

(continues on next page)

(continued from previous page)

```
  institute: MRI
- dataset: MRI-ESM2-0
  ensemble: r1i1000p1f1
  grid: gn
  institute: MRI
- dataset: MRI-ESM2-0
  ensemble: r(1:10)i1p1f1
  grid: gn
  institute: MRI
- dataset: GISS-E2-1-G-CC
  ensemble: r1i1p1f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(1:4)i1p5f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(1:5)i1p1f3
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(1:10)i1p1f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(1:10)i1p3f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(1:11)i1p1f2
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(6:10)i1p5f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-G
  ensemble: r(101:102)i1p1f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-H
  ensemble: r(1:5)i1p1f2
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-H
  ensemble: r(1:5)i1p3f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-1-H
  ensemble: r(1:5)i1p5f1
  grid: gn
```

(continues on next page)

(continued from previous page)

```
  institute: NASA-GISS
- dataset: GISS-E2-1-H
  ensemble: r(1:10)i1p1f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-2-G
  ensemble: r(1:5)i1p3f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-2-G
  ensemble: r(1:6)i1p1f1
  grid: gn
  institute: NASA-GISS
- dataset: GISS-E2-2-H
  ensemble: r(1:5)i1p1f1
  grid: gn
  institute: NASA-GISS
- dataset: CESM2-FV2
  ensemble: r1i2p2f1
  grid: gn
  institute: NCAR
- dataset: CESM2-FV2
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: NCAR
- dataset: CESM2-WACCM-FV2
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: NCAR
- dataset: CESM2-WACCM
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: NCAR
- dataset: CESM2
  ensemble: r(1:11)i1p1f1
  grid: gn
  institute: NCAR
- dataset: NorCPM1
  ensemble: r(1:30)i1p1f1
  grid: gn
  institute: NCC
- dataset: NorESM2-LM
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: NCC
- dataset: NorESM2-MM
  ensemble: r(1:3)i1p1f1
  grid: gn
  institute: NCC
- dataset: KACE-1-0-G
  ensemble: r(1:3)i1p1f1
  grid: gr
```

(continues on next page)

(continued from previous page)

```
  institute: NIMS-KMA
- dataset: UKESM1-0-LL
  ensemble: r(13:15)i1p1f2
  grid: gn
  institute: NIMS-KMA
- dataset: GFDL-CM4
  ensemble: r1i1p1f1
  grid: gr1
  institute: NOAA-GFDL
- dataset: GFDL-ESM4
  ensemble: r(1:3)i1p1f1
  grid: gr1
  institute: NOAA-GFDL
- dataset: NESM3
  ensemble: r(1:5)i1p1f1
  grid: gn
  institute: NUIST
- dataset: SAM0-UNICON
  ensemble: r1i1p1f1
  grid: gn
  institute: SNU
- dataset: CIESM
  ensemble: r(1:3)i1p1f1
  grid: gr
  institute: THU
- dataset: MCM-UA-1-0
  ensemble: r1i1p1f(1:2)
  grid: gn
  institute: UA
diagnostics:
  diagnostic_name:
    variables:
      tas:
        exp: historical
        mip: Amon
        project: CMIP6
```

CHAPTER

SEVEN

DISCOVERING DATA

This notebook shows how to find out what data is available locally as well as on ESGF. It also shows how to download the data from ESGF.

[1]:
```
from esmvalcore.config import CFG
from esmvalcore.dataset import Dataset, datasets_to_recipe
from esmvalcore.esgf import download
import yaml
```

Configure ESMValCore so it always searches the ESGF for data

[2]:
```
CFG['search_esgf'] = 'always'
```

We define a dataset template to search for all CMIP6 datasets that provide surface air temperature (tas) on a monthly resolution for the historical experiment. Note that ESMValCore uses its own names for the facets for a more uniform naming across different CMIP phases and other projects. The mapping to the facet names used on ESGF can be found in esmvalcore.esgf.facets.FACETS.

[3]:
```
dataset_template = Dataset(
    short_name='tas',
    mip='Amon',
    project='CMIP6',
    exp='historical',
    dataset='*',
    institute='*',
    ensemble='*',
    grid='*',
)
```

Next, we use the `Dataset.from_files` method to build a list of datasets from the available files. This may take a while as searching the ESGF for many files is a bit slow. Because the search results are cached for a configurable duration, subsequent searches will be faster.

[4]:
```
datasets = list(dataset_template.from_files())
print(f"Found {len(datasets)} datasets, showing the first 10:")
datasets[:10]
```

```
Found 778 datasets, showing the first 10:
```

[4]:
```
[Dataset:
 {'dataset': 'TaiESM1',
  'project': 'CMIP6',
  'mip': 'Amon',
```

(continues on next page)

(continued from previous page)

```
 'short_name': 'tas',
 'ensemble': 'r1i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AS-RCEC'},
Dataset:
{'dataset': 'TaiESM1',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r2i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AS-RCEC'},
Dataset:
{'dataset': 'AWI-CM-1-1-MR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r1i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AWI'},
Dataset:
{'dataset': 'AWI-CM-1-1-MR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r2i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AWI'},
Dataset:
{'dataset': 'AWI-CM-1-1-MR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r3i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AWI'},
Dataset:
{'dataset': 'AWI-CM-1-1-MR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r4i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AWI'},
Dataset:
{'dataset': 'AWI-CM-1-1-MR',
```

(continues on next page)

(continued from previous page)

```
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r5i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AWI'},
Dataset:
{'dataset': 'AWI-ESM-1-1-LR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r1i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AWI'},
Dataset:
{'dataset': 'BCC-CSM2-MR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r1i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'BCC'},
Dataset:
{'dataset': 'BCC-CSM2-MR',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r2i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'BCC'}]
```

Let's look at the first dataset in more detail. We can print the facets describing the dataset:

[5]:
```
dataset = datasets[0]
dataset
```

[5]:
```
Dataset:
{'dataset': 'TaiESM1',
 'project': 'CMIP6',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r1i1p1f1',
 'exp': 'historical',
 'grid': 'gn',
 'institute': 'AS-RCEC'}
```

and see what files are available:

[6]:
```
dataset.files
```

[6]:
```
[ESGFFile:CMIP6/CMIP/AS-RCEC/TaiESM1/historical/r1i1p1f1/Amon/tas/gn/v20200623/tas_Amon_
↪TaiESM1_historical_r1i1p1f1_gn_185001-201412.nc on hosts ['esgf-data1.llnl.gov', 'esgf.
↪ceda.ac.uk', 'esgf.rcec.sinica.edu.tw', 'esgf3.dkrz.de', 'esgf-data04.diasjp.net',
↪'esgf.nci.org.au', 'esgf3.dkrz.de']]
```

A single file can be downloaded using its `download` method:

[7]:
```
dataset.files[0].download(CFG['download_dir'])
```

[7]:
```
LocalFile('~/climate_data/CMIP6/CMIP/AS-RCEC/TaiESM1/historical/r1i1p1f1/Amon/tas/gn/
↪v20200623/tas_Amon_TaiESM1_historical_r1i1p1f1_gn_185001-201412.nc')
```

For downloading many files, the esmvalcore.esgf.download function is recommended because it will download the files in parallel. The ESMValCore will try to guess the fastest host and download from there. If it is not available for some reason, it will automatically fall back to the next host.

[8]:
```
download(dataset.files, CFG['download_dir'])
```

CHAPTER

EIGHT

LOADING, PROCESSING, AND VISUALIZING DATA

This notebook shows how to load a dataset, use the preprocessor functions, and visualize the result. In this notebook we will plot the annual mean temperature from 1850 till 2100 from one model.

[1]:
```
%matplotlib inline

import matplotlib.pyplot as plt
import iris.quickplot

from esmvalcore.config import CFG
from esmvalcore.dataset import Dataset
from esmvalcore.esgf import download, ESGFFile
from esmvalcore.preprocessor import area_statistics, annual_statistics
```

Configure ESMValCore so it searches the ESGF for data

[2]:
```
CFG['search_esgf'] = 'when_missing'
```

Define the dataset we are going to use. In this case surface air temperature (`tas`).

[3]:
```
tas = Dataset(
    short_name='tas',
    mip='Amon',
    project='CMIP5',
    dataset='MPI-ESM-MR',
    ensemble='r1i1p1',
    exp='historical',
    timerange='1850/2000',
)
tas
```

[3]:
```
Dataset:
{'dataset': 'MPI-ESM-MR',
 'project': 'CMIP5',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r1i1p1',
 'exp': 'historical',
 'timerange': '1850/2000'}
```

In order to compute the area average later on, we add the cell areas (`areacella`) as a supplementary dataset. This will append a new dataset to the list of supplementary datasets. Its facets are copied from the `tas` dataset and updated with the provided facets:

[4]:
```
tas.add_supplementary(short_name='areacella', mip='fx', ensemble='r0i0p0')
tas.supplementaries
```

[4]:
```
[Dataset:
 {'dataset': 'MPI-ESM-MR',
  'project': 'CMIP5',
  'mip': 'fx',
  'short_name': 'areacella',
  'ensemble': 'r0i0p0',
  'exp': 'historical',
  'timerange': '1850/2000'}]
```

ESMValCore can automatically add extra facets based on the `project`, `mip`, `short_name`, and `dataset`. These extra facets are automatically added and used when searching for input files.

[5]:
```
tas.augment_facets()
tas
```

[5]:
```
Dataset:
{'dataset': 'MPI-ESM-MR',
 'project': 'CMIP5',
 'mip': 'Amon',
 'short_name': 'tas',
 'ensemble': 'r1i1p1',
 'exp': 'historical',
 'frequency': 'mon',
 'institute': ['MPI-M'],
 'long_name': 'Near-Surface Air Temperature',
 'modeling_realm': ['atmos'],
 'original_short_name': 'tas',
 'product': ['output1', 'output2'],
 'standard_name': 'air_temperature',
 'timerange': '1850/2000',
 'units': 'K'}
supplementaries:
  {'dataset': 'MPI-ESM-MR',
   'project': 'CMIP5',
   'mip': 'fx',
   'short_name': 'areacella',
   'ensemble': 'r0i0p0',
   'exp': 'historical',
   'frequency': 'fx',
   'institute': ['MPI-M'],
   'long_name': 'Atmosphere Grid-Cell Area',
   'modeling_realm': ['atmos', 'land'],
   'original_short_name': 'areacella',
   'product': ['output1', 'output2'],
   'standard_name': 'cell_area',
   'units': 'm2'}
session: 'session-686367c0-001c-4864-839d-c20887cf7415_20230301_160531'
```

Use the `find_files` method to find the files corresponding to the dataset.

[6]:
```
tas.find_files()
```

(continues on next page)

(continued from previous page)

```
print(tas.files)
for supplementary_ds in tas.supplementaries:
    print(supplementary_ds.files)
```

```
[ESGFFile:cmip5/output1/MPI-M/MPI-ESM-MR/historical/mon/atmos/Amon/r1i1p1/v20120503/tas_
↪Amon_MPI-ESM-MR_historical_r1i1p1_185001-200512.nc on hosts ['aims3.llnl.gov', 'esgf.
↪ceda.ac.uk', 'esgf.nci.org.au', 'esgf1.dkrz.de']]
[ESGFFile:cmip5/output1/MPI-M/MPI-ESM-MR/historical/fx/atmos/fx/r0i0p0/v20120503/
↪areacella_fx_MPI-ESM-MR_historical_r0i0p0.nc on hosts ['aims3.llnl.gov', 'esgf.ceda.ac.
↪uk', 'esgf.nci.org.au', 'esgf1.dkrz.de']]
```

If the files are not available locally, ESMValCore can download them for us.

[7]:
```
files = list(tas.files)
for supplementary_ds in tas.supplementaries:
    files.extend(supplementary_ds.files)
files = [file for file in files if isinstance(file, ESGFFile)]
download(files, CFG['download_dir'])
tas.find_files()
print(tas.files)
for supplementary_ds in tas.supplementaries:
    print(supplementary_ds.files)
```

```
[LocalFile('~/climate_data/cmip5/output1/MPI-M/MPI-ESM-MR/historical/mon/atmos/Amon/
↪r1i1p1/v20120503/tas_Amon_MPI-ESM-MR_historical_r1i1p1_185001-200512.nc')]
[LocalFile('~/climate_data/cmip5/output1/MPI-M/MPI-ESM-MR/historical/fx/atmos/fx/r0i0p0/
↪v20120503/areacella_fx_MPI-ESM-MR_historical_r0i0p0.nc')]
```

The data in the files can be loaded into an iris.cube.Cube. ESMValCore will automatically check for and fix problems with the data formatting and attach the cell area.

[8]:
```
cube = tas.load()
cube
```

[8]:
```
<iris 'Cube' of air_temperature / (K) (time: 1812; latitude: 96; longitude: 192)>
```

[9]:
```
cell_area = cube.cell_measures()[0]
cell_area
```

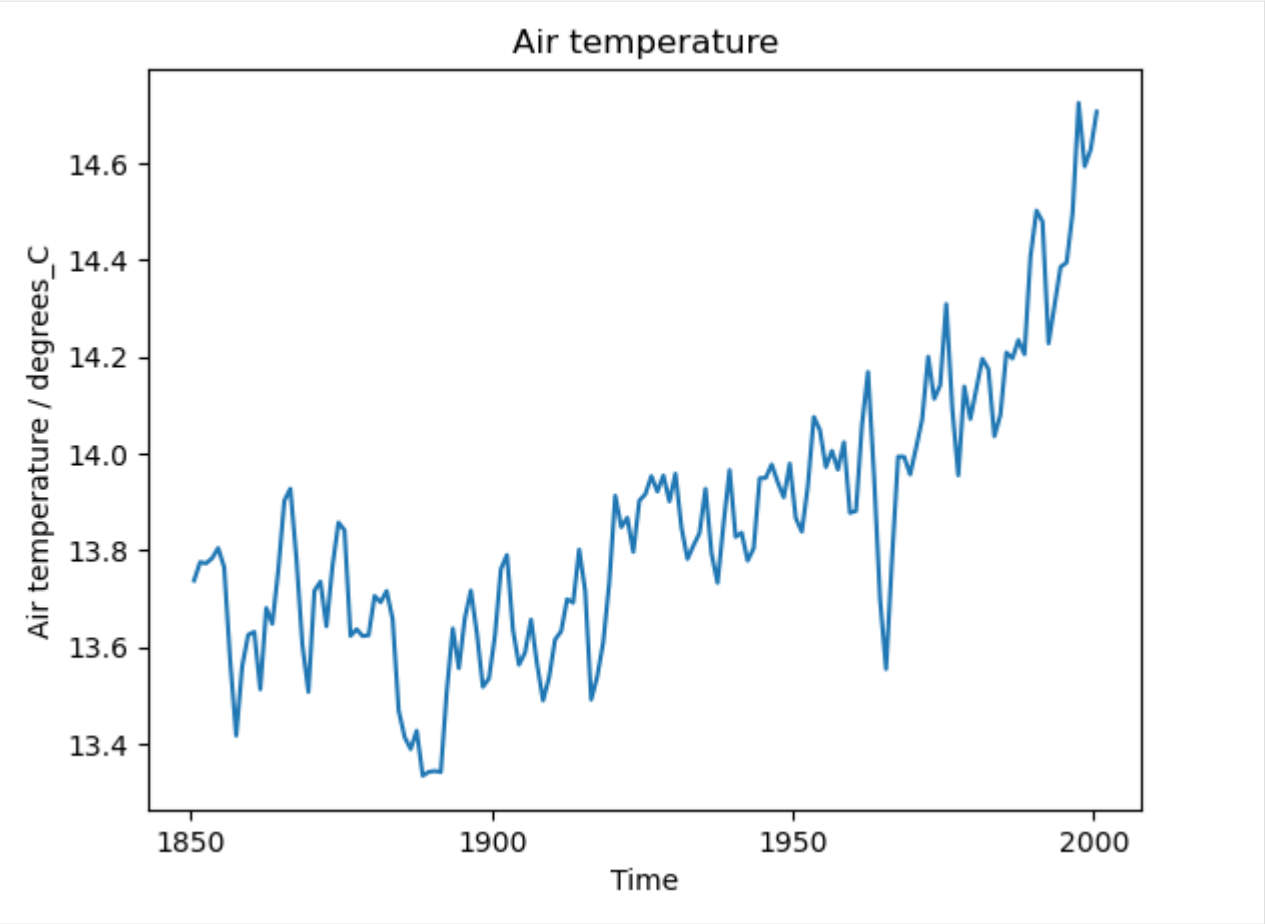
[9]:
```
<CellMeasure: cell_area / (m2)  <lazy>  shape(96, 192)>
```

This code shows how to use some esmvalcore.preprocessor functions to compute the global annual mean temperature in degrees Celsius:

[10]:
```
cube = area_statistics(cube, operator='mean')
cube = annual_statistics(cube, operator='mean')
cube.convert_units('degrees_C')
```

The iris.quickplot module has useful functions for quickly plotting the results:

[11]:
```
iris.quickplot.plot(cube)
plt.show()
```


Air temperature
Air temperature / degrees_C
14.6
14.4
14.2
14.0
13.8
13.6
13.4
1850
1900
1950
2000
Time

Part III

The recipe format

CHAPTER

NINE

OVERVIEW

After `config-user.yml`, the `recipe.yml` is the second file the user needs to pass to `esmvaltool` as command line option, at each run time point. Recipes contain the data and data analysis information and instructions needed to run the diagnostic(s), as well as specific diagnostic-related instructions.

Broadly, recipes contain a general section summarizing the provenance and functionality of the diagnostics, the datasets which need to be run, the preprocessors that need to be applied, and the diagnostics which need to be run over the preprocessed data. This information is provided to ESMValTool in four main recipe sections: *Documentation*, *Datasets*, *Preprocessors*, and *Diagnostics*, respectively.

9.1 Recipe section: `documentation`

The documentation section includes:

- The recipe's author's user name (`authors`, matching the definitions in the *References configuration file*)
- The recipe's maintainer's user name (`maintainer`, matching the definitions in the *References configuration file*)
- The title of the recipe (`title`)
- A description of the recipe (`description`, written in MarkDown format)
- A list of scientific references (`references`, matching the definitions in the *References configuration file*)
- the project or projects associated with the recipe (`projects`, matching the definitions in the *References configuration file*)

For example, the documentation section of `recipes/recipe_ocean_amoc.yml` is the following:

```
documentation:
  title: Atlantic Meridional Overturning Circulation (AMOC) and the drake passage current
  description: |
    Recipe to produce time series figures of the derived variable, the
    Atlantic meridional overturning circulation (AMOC).
    This recipe also produces transect figures of the stream functions for
    the years 2001-2004.

  authors:
    - demo_le

  maintainer:
    - demo_le

  references:
```

(continues on next page)

(continued from previous page)

```
    - demora2018gmd

  projects:
    - ukesm
```

Note: Note that all authors, projects, and references mentioned in the description section of the recipe need to be included in the (locally installed copy of the) file esmvaltool/config-references.yml, see *References configuration file*. The author name uses the format: `surname_name`. For instance, John Doe would be: `doe_john`. This information can be omitted by new users whose name is not yet included in `config-references.yml`.

9.2 Recipe section: `datasets`

The `datasets` section includes dictionaries that, via key-value pairs or "facets", define standardized data specifications:

- dataset name (key `dataset`, value e.g. `MPI-ESM-LR` or `UKESM1-0-LL`).
- project (key `project`, value `CMIP5` or `CMIP6` for CMIP data, `OBS` for observational data, `ana4mips` for ana4mips data, `obs4MIPs` for obs4MIPs data, `ICON` for ICON data).
- experiment (key `exp`, value e.g. `historical`, `amip`, `piControl`, `rcp85`).
- mip (for CMIP data, key `mip`, value e.g. `Amon`, `Omon`, `LImon`).
- ensemble member (key `ensemble`, value e.g. `r1i1p1`, `r1i1p1f1`).
- sub-experiment id (key `sub_experiment`, value e.g. `s2000`, `s(2000:2002)`, for DCPP data only).
- time range (e.g. key-value `start_year: 1982`, `end_year: 1990`). Please note that yaml interprets numbers with a leading `0` as octal numbers, so we recommend to avoid them. For example, use `128` to specify the year 128 instead of `0128`. Alternatively, the time range can be specified in ISO 8601 format, for both dates and periods. In addition, wildcards (`'*'`) are accepted, which allow the selection of the first available year for each individual dataset (when used as a starting point) or the last available year (when used as an ending point). The starting point and end point must be separated with `/` (e.g. key-value `timerange: '1982/1990'`). More examples are given *here*.
- model grid (native grid `grid: gn` or regridded grid `grid: gr`, for CMIP6 data only).

For example, a datasets section could be:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: r1i1p1, start_year:␣
↪2001, end_year: 2004}
  - {dataset: UKESM1-0-LL, project: CMIP6, exp: historical, ensemble: r1i1p1f2, start_
↪year: 2001, end_year: 2004, grid: gn}
  - {dataset: ACCESS-CM2, project: CMIP6, exp: historical, ensemble: r1i1p1f2, timerange:
↪ 'P5Y/*', grid: gn}
  - {dataset: EC-EARTH3, alias: custom_alias, project: CMIP6, exp: historical, ensemble:␣
↪r1i1p1f1, start_year: 2001, end_year: 2004, grid: gn}
  - {dataset: CMCC-CM2-SR5, project: CMIP6, exp: historical, ensemble: r1i1p1f1,␣
↪timerange: '2001/P10Y', grid: gn}
  - {dataset: HadGEM3-GC31-MM, project: CMIP6, exp: dcppA-hindcast, ensemble: r1i1p1f1,␣
↪sub_experiment: s2000, grid: gn, start_year: 2000, end_year, 2002}
```

(continues on next page)

(continued from previous page)

```
  - {dataset: BCC-CSM2-MR, project: CMIP6, exp: dcppA-hindcast, ensemble: r1i1p1f1, sub_
↪experiment: s2000, grid: gn, timerange: '*'}
```

9.2.1 Automatically populating a recipe with all available datasets

It is possible to use `glob` patterns or wildcards for certain facet values, to make it easy to find all available datasets locally and/or on ESGF. Note that `project` cannot be a wildcard.

The facet values for local files are retrieved from the directory tree where the directories represent the facets values. Reading facet values from file names is not yet supported. See *CMIP data* for more information on this kind of file organization.

When (some) files are available locally, the tool will not automatically look for more files on ESGF. To populate a recipe with all available datasets from ESGF, `search_esgf` should be set to `always` in the *user configuration file*.

For more control over which datasets are selected, it is recommended to use a Python script or Jupyter notebook to compose the recipe. See *Composing recipes* for an example. This is particularly useful when specific relations are required between datasets, e.g. when a dataset needs to be available for multiple variables or experiments.

An example recipe that will use all CMIP6 datasets and all ensemble members which have a `'historical'` experiment could look like this:

```
datasets:
  - project: CMIP6
    exp: historical
    dataset: '*'
    institute: '*'
    ensemble: '*'
    grid: '*'
```

After running the recipe, a copy specifying exactly which datasets were used is available in the output directory in the `run` subdirectory. The filename of this recipe will end with `_filled.yml`.

For the `timerange` facet, special syntax is available. See *Time ranges* for more information.

If populating a recipe using wildcards does not work, this is because there were either no files found that match those facets, or the facets could not be read from the directory name or ESGF.

9.2.2 Defining supplementary variables (ancillary variables and cell measures)

It is common practice to store ancillary variables (e.g. land/sea/ice masks) and cell measures (e.g. cell area, cell volume) in separate datasets that are described by slightly different facets. In ESMValCore, we call ancillary variables and cell measures "supplementary variables". Some *preprocessor functions* need this information to work. For example, the *area_statistics* preprocessor function needs to know area of each grid cell in order to compute a correctly weighted statistic.

To attach these variables to a dataset, the `supplementary_variables` keyword can be used. For example, to add cell area to a dataset, it can be specified as follows:

```
datasets:
  - dataset: BCC-ESM1
    project: CMIP6
    exp: historical
    ensemble: r1i1p1f1
```

(continues on next page)

(continued from previous page)

```
    grid: gn
    supplementary_variables:
      - short_name: areacella
        mip: fx
        exp: 1pctCO2
```

Note that the supplementary variable will inherit the facet values from the main dataset, so only those facet values that differ need to be specified.

9.2.3 Automatically selecting the supplementary dataset

When using many datasets, it may be quite a bit of work to find out which facet values are required to find the corresponding supplementary data. The tool can automatically guess the best matching supplementary dataset. To use this feature, the supplementary dataset can be specified as:

```
datasets:
  - dataset: BCC-ESM1
    project: CMIP6
    exp: historical
    ensemble: r1i1p1f1
    grid: gn
    supplementary_variables:
      - short_name: areacella
        mip: fx
        exp: '*'
        activity: '*'
        ensemble: '*'
```

With this syntax, the tool will search all available values of `exp`, `activity`, and `ensemble` and use the supplementary dataset that shares the most facet values with the main dataset. Note that this behaviour is different from *using wildcards in the main dataset*, where they will be expanded to generate all matching datasets. The available datasets are shown in the debug log messages when running a recipe with wildcards, so if a different supplementary dataset is preferred, these messages can be used to see what facet values are available. The facet values for local files are retrieved from the directory tree where the directories represent the facets values. Reading facet values from file names is not yet supported. If wildcard expansion fails, this is because there were either no files found that match those facets, or the facets could not be read from the directory name or ESGF.

9.2.4 Automatic definition of supplementary variables

If an ancillary variable or cell measure is *needed by a preprocessor function*, but it is not specified in the recipe, the tool will automatically make a best guess using the syntax above. Usually this will work fine, but if it does not, it is recommended to explicitly define the supplementary variables in the recipe.

To disable this automatic addition, define the supplementary variable as usual, but add the special facet `skip` with value `true`. See *Supplementary variables (ancillary variables and cell measures)* for an example recipe.

9.2.5 Saving ancillary variables and cell measures

By default, ancillary variables and cell measures will be removed from the main variable before saving it to file because they can be as big as the main variable. To keep the supplementary variables, disable the preprocessor function that removes them by setting `remove_supplementary_variables: false` in the preprocessor profile in the recipe.

9.2.6 Concatenating data corresponding to multiple facets

It is possible to define the experiment as a list to concatenate two experiments. Here it is an example concatenating the *historical* experiment with *rcp85*

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: [historical, rcp85], ensemble: r1i1p1, start_
↪year: 2001, end_year: 2004}
```

It is also possible to define the ensemble as a list when the two experiments have different ensemble names. In this case, the specified datasets are concatenated into a single cube:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: [historical, rcp85], ensemble: [r1i1p1,␣
↪r1i2p1], start_year: 2001, end_year: 2004}
```

9.2.7 Short notation of ensemble members and sub-experiments

ESMValTool also supports a simplified syntax to add multiple ensemble members from the same dataset. In the ensemble key, any element in the form *(x:y)* will be replaced with all numbers from x to y (both inclusive), adding a dataset entry for each replacement. For example, to add ensemble members r1i1p1 to r10i1p1 you can use the following abbreviated syntax:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: "r(1:10)i1p1", start_
↪year: 2001, end_year: 2004}
```

It can be included multiple times in one definition. For example, to generate the datasets definitions for the ensemble members r1i1p1 to r5i1p1 and from r1i2p1 to r5i1p1 you can use:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: "r(1:5)i(1:2)p1",␣
↪start_year: 2001, end_year: 2004}
```

Please, bear in mind that this syntax can only be used in the ensemble tag. Also, note that the combination of multiple experiments and ensembles, like exp: [historical, rcp85], ensemble: [r1i1p1, "r(2:3)i1p1"] is not supported and will raise an error.

The same simplified syntax can be used to add multiple sub-experiments:

```
datasets:
  - {dataset: MIROC6, project: CMIP6, exp: dcppA-hindcast, ensemble: r1i1p1f1, sub_
↪experiment: s(2000:2002), grid: gn, start_year: 2003, end_year: 2004}
```

9.2.8 Time ranges

When using the `timerange` tag to specify the start and end points, possible values can be as follows:

timerange	effect
`'1980/1982'`	Spans from 01/01/1980 to 31/12/1982
`'198002/198205'`	Spans from 01/02/1980 to 31/05/1982
`'19800302/19820403'`	Spans from 02/03/1980 to 03/04/1982
`'19800504T100000/19800504T110000'`	Spans from 04/05/1980 at 10h to 11h
`'1980/P5Y'`	Starting from 01/01/1980, spans 5 years
`'P2Y5M/198202`	Ending at 28/02/1982, spans 2 years and 5 months
`'*'`	Finds all available years
`'*/1982`	Finds first available point, spans to 31/12/1982
`'*/P6Y`	Finds first available point, spans 6 years from it
`'198003/*`	Starting from 01/03/1980, spans until the last available point
`'P5M/*`	Finds last available point, spans 5 months backwards from it

Note: Please make sure to use a consistent number of digits for the start and end point when using `timerange`, e.g., instead of `198005/2000`, use `198005/200012`. Otherwise, it might happen that ESMValTool does not find your data even though the corresponding years are available. This also applies to wildcards: Wildcards are usually resolved using the timerange in the file name. If this is given in the form `YYYYMM`, then the other time point in `timerange` needs to be in the same format, e.g., use `*/200012` instead of `*/2000` in this case. If you use wildcards and get an unexpected error about missing data, have a look at the resolved `timerange` in the error message (`ERROR No input files found for variable {'timerange': '197901/2000', ...}`) and make sure that the number of digits in it is consistent.

Note that this section is not required, as datasets can also be provided in the *Diagnostics* section.

9.3 Recipe section: `preprocessors`

The preprocessor section of the recipe includes one or more preprocessors, each of which may call the execution of one or several preprocessor functions.

Each preprocessor section includes:

- A preprocessor name (any name, under `preprocessors`);
- A list of preprocessor steps to be executed (choose from the API);
- Any or none arguments given to the preprocessor steps;
- The order that the preprocessor steps are applied can also be specified using the `custom_order` preprocessor function.

The following snippet is an example of a preprocessor named `prep_map` that contains multiple preprocessing steps (*Horizontal regridding* with two arguments, *Time manipulation* with no arguments (i.e., calculating the average over the time dimension) and *Multi-model statistics* with two arguments):

```
preprocessors:
  prep_map:
    regrid:
      target_grid: 1x1
```

(continues on next page)

(continued from previous page)

```
      scheme: linear
    climate_statistics:
      operator: mean
    multi_model_statistics:
      span: overlap
      statistics: [mean]
```

Note: In this case no `preprocessors` section is needed the workflow will apply a `default` preprocessor consisting of only basic operations like: loading data, applying CMOR checks and fixes (*CMORization and dataset-specific fixes*) and saving the data to disk.

Preprocessor operations will be applied using the default order as listed in *Preprocessor functions*. Preprocessor tasks can be set to run in the order they are listed in the recipe by adding `custom_order: true` to the preprocessor definition.

9.4 Recipe section: `diagnostics`

The diagnostics section includes one or more diagnostics. Each diagnostic section will include:

- the variable(s) to preprocess, including the preprocessor to be applied to each variable;
- the diagnostic script(s) to be run;
- a description of the diagnostic and lists of themes and realms that it applies to;
- an optional `additional_datasets` section.
- an optional `title` and `description`, used to generate the title and description in the `index.html` output file.

9.4.1 The diagnostics section defines tasks

The diagnostic section(s) define the tasks that will be executed when running the recipe. For each variable a preprocessing task will be defined and for each diagnostic script a diagnostic task will be defined. These tasks can be viewed in the main_log_debug.txt file that is produced every run. Each task has a unique name that defines the subdirectory where the results of that task are stored. Task names start with the name of the diagnostic section followed by a '/' and then the name of the variable section for a preprocessing task or the name of the diagnostic script section for a diagnostic task.

A (simplified) example diagnostics section could look like

```
diagnostics:
  diagnostic_name:
    title: Air temperature tutorial diagnostic
    description: A longer description can be added here.
    themes:
      - phys
    realms:
      - atmos
    variables:
      variable_name:
        short_name: ta
        preprocessor: preprocessor_name
```

(continues on next page)

(continued from previous page)

```
      mip: Amon
  scripts:
    script_name:
      script: examples/diagnostic.py
```

Note that the example recipe above contains a single diagnostic section called `diagnostic_name` and will result in two tasks:

- a preprocessing task called `diagnostic_name/variable_name` that will preprocess air temperature data for each dataset in the *Datasets* section of the recipe (not shown).
- a diagnostic task called `diagnostic_name/script_name`

The path to the script provided in the `script` option should be either the absolute path to the script, or the path relative to the `esmvaltool/diag_scripts` directory.

Depending on the installation configuration, you may get an error of "file does not exist" when the system tries to run the diagnostic script using relative paths. If this happens, use an absolute path instead.

Note that the script should either have the extension for a supported language, i.e. `.py`, `.R`, `.ncl`, or `.jl` for Python, R, NCL, and Julia diagnostics respectively, or be executable if it is written in any other language.

9.4.2 Ancestor tasks

Some tasks require the result of other tasks to be ready before they can start, e.g. a diagnostic script needs the preprocessed variable data to start. Thus each tasks has zero or more ancestor tasks. By default, each diagnostic task in a diagnostic section has all variable preprocessing tasks in that same section as ancestors. However, this can be changed using the `ancestors` keyword. Note that wildcard expansion can be used to define ancestors.

```
diagnostics:
  diagnostic_1:
    variables:
      airtemp:
        short_name: ta
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_a:
        script: diagnostic_a.py
  diagnostic_2:
    variables:
      precip:
        short_name: pr
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_b:
        script: diagnostic_b.py
        ancestors: [diagnostic_1/script_a, precip]
```

The example recipe above will result in four tasks:

- a preprocessing task called `diagnostic_1/airtemp`
- a diagnostic task called `diagnostic_1/script_a`

- a preprocessing task called `diagnostic_2/precip`
- a diagnostic task called `diagnostic_2/script_b`

the preprocessing tasks do not have any ancestors, while the diagnostic_a.py script will receive the preprocessed air temperature data (has ancestor `diagnostic_1/airtemp`) and the diagnostic_b.py script will receive the results of diagnostic_a.py and the preprocessed precipitation data (has ancestors `diagnostic_1/script_a` and `diagnostic_2/precip`).

9.4.3 Task priority

Tasks are assigned a priority, with tasks appearing earlier on in the recipe getting higher priority. The tasks will be executed sequentially or in parallel, depending on the setting of `max_parallel_tasks` in the *User configuration file*. When there are fewer than `max_parallel_tasks` running, tasks will be started according to their priority. For obvious reasons, only tasks that are not waiting for ancestor tasks can be started. This feature makes it possible to reduce the processing time of recipes with many tasks, by placing tasks that take relatively long near the top of the recipe. Of course this only works when settings `max_parallel_tasks` to a value larger than 1. The current priority and run time of individual tasks can be seen in the log messages shown when running the tool (a lower number means higher priority).

9.4.4 Variable and dataset definitions

To define a variable/dataset combination that corresponds to an actual variable from a dataset, the keys in each variable section are combined with the keys of each dataset definition. If two versions of the same key are provided, then the key in the datasets section will take precedence over the keys in variables section. For many recipes it makes more sense to define the `start_year` and `end_year` items in the variable section, because the diagnostic script assumes that all the data has the same time range.

Variable short names usually do not change between datasets supported by ESMValCore, as they are usually changed to match CMIP. Nevertheless, there are small changes in variable names in CMIP6 with respect to CMIP5 (i.e. sea ice concentration changed from `sic` to `siconc`). ESMValCore is aware of some of them and can do the automatic translation when needed. It will even do the translation in the preprocessed file so the diagnostic does not have to deal with this complexity, setting the short name in all files to match the one used by the recipe. For example, if `sic` is requested, ESMValCore will find `sic` or `siconc` depending on the project, but all preprocessed files while use `sic` as their short_name. If the recipe requested `siconc`, the preprocessed files will be identical except that they will use the short_name `siconc` instead.

9.4.5 Diagnostic and variable specific datasets

The `additional_datasets` option can be used to add datasets beyond those listed in the *Datasets* section. This is useful if specific datasets need to be used only by a specific diagnostic or variable, i.e. it can be added both at diagnostic level, where it will apply to all variables in that diagnostic section or at individual variable level. For example, this can be a good way to add observational datasets, which are usually variable-specific.

9.4.6 Running a simple diagnostic

The following example, taken from `recipe_ocean_example.yml`, shows a diagnostic named *diag_map*, which loads the temperature at the ocean surface between the years 2001 and 2003 and then passes it to the `prep_map` preprocessor. The result of this process is then passed to the ocean diagnostic map script, `ocean/diagnostic_maps.py`.

```
diagnostics:

  diag_map:
    title: Global Ocean Surface regridded temperature map
    description: Add a longer description here.
    variables:
      tos: # Temperature at the ocean surface
        preprocessor: prep_map
        start_year: 2001
        end_year: 2003
    scripts:
      Global_Ocean_Surface_regrid_map:
        script: ocean/diagnostic_maps.py
```

9.4.7 Passing arguments to a diagnostic script

The diagnostic script section(s) may include custom arguments that can be used by the diagnostic script; these arguments are stored at runtime in a dictionary that is then made available to the diagnostic script via the interface link, independent of the language the diagnostic script is written in. Here is an example of such groups of arguments:

```
scripts:
  autoassess_strato_test_1: &autoassess_strato_test_1_settings
    script: autoassess/autoassess_area_base.py
    title: "Autoassess Stratosphere Diagnostic Metric MPI-MPI"
    area: stratosphere
    control_model: MPI-ESM-LR
    exp_model: MPI-ESM-MR
    obs_models: [ERA-Interim]  # list to hold models that are NOT for metrics but for␣
↪obs operations
    additional_metrics: [ERA-Interim, inmcm4]  # list to hold additional datasets for␣
↪metrics
```

In this example, apart from specifying the diagnostic script `script: autoassess/autoassess_area_base.py`, we pass a suite of parameters to be used by the script (`area`, `control_model` etc). These parameters are stored in key-value pairs in the diagnostic configuration file, an interface file that can be used by importing the `run_diagnostic` utility:

```
from esmvaltool.diag_scripts.shared import run_diagnostic

# write the diagnostic code here e.g.
def run_some_diagnostic(my_area, my_control_model, my_exp_model):
    """Diagnostic to be run."""
    if my_area == 'stratosphere':
        diag = my_control_model / my_exp_model
        return diag
```

(continues on next page)

(continued from previous page)

```
def main(cfg):
    """Main diagnostic run function."""
    my_area = cfg['area']
    my_control_model = cfg['control_model']
    my_exp_model = cfg['exp_model']
    run_some_diagnostic(my_area, my_control_model, my_exp_model)

if __name__ == '__main__':

    with run_diagnostic() as config:
        main(config)
```

This way a lot of the optional arguments necessary to a diagnostic are at the user's control via the recipe.

9.4.8 Running your own diagnostic

If the user wants to test a newly-developed `my_first_diagnostic.py` which is not yet part of the ESMValTool diagnostics library, he/she do it by passing the absolute path to the diagnostic:

```
diagnostics:

  myFirstDiag:
    title: Let's do some science!
    description: John Doe wrote a funny diagnostic
    variables:
      tos: # Temperature at the ocean surface
        preprocessor: prep_map
        start_year: 2001
        end_year: 2003
    scripts:
      JoeDiagFunny:
        script: /home/users/john_doe/esmvaltool_testing/my_first_diagnostic.py
```

This way the user may test a new diagnostic thoroughly before committing to the GitHub repository and including it in the ESMValTool diagnostics library.

9.4.9 Re-using parameters from one `script` to another

Due to `yaml` features it is possible to recycle entire diagnostics sections for use with other diagnostics. Here is an example:

```
scripts:
  cycle: &cycle_settings
    script: perfmetrics/main.ncl
    plot_type: cycle
    time_avg: monthlyclim
  grading: &grading_settings
    <<: *cycle_settings
    plot_type: cycle_latlon
    calc_grading: true
    normalization: [centered_median, none]
```

In this example the hook `&cycle_settings` can be used to pass the `cycle:` parameters to `grading:` via the shortcut `<<: *cycle_settings`.

CHAPTER

TEN

PREPROCESSOR

In this section, each of the preprocessor modules is described, roughly following the default order in which preprocessor functions are applied:

- *Overview*
- *Statistical preprocessors*
- *Variable derivation*
- *CMORization and dataset-specific fixes*
- *Supplementary variables (ancillary variables and cell measures)*
- *Vertical interpolation*
- *Weighting*
- *Land-sea masking*
- *Horizontal regridding*
- *Missing values masks*
- *Ensemble statistics*
- *Multi-model statistics*
- *Time manipulation*
- *Area manipulation*
- *Volume manipulation*
- *Cycles*
- *Trend*
- *Detrend*
- *Rolling window statistics*
- *Unit conversion*
- *Bias*
- *Other*

See *Preprocessor functions* for implementation details and the exact default order.

10.1 Overview

The ESMValCore preprocessor can be used to perform a broad range of operations on the input data before diagnostics or metrics are applied. The preprocessor performs these operations in a centralized, documented and efficient way, thus reducing the data processing load on the diagnostics side. For an overview of the preprocessor structure see the *Recipe section: preprocessors*.

Each of the preprocessor operations is written in a dedicated python module and all of them receive and return an instance of `iris.cube.Cube`, working sequentially on the data with no interactions between them. The order in which the preprocessor operations is applied is set by default to minimize the loss of information due to, for example, temporal and spatial subsetting or multi-model averaging. Nevertheless, the user is free to change such order to address specific scientific requirements, but keeping in mind that some operations must be necessarily performed in a specific order. This is the case, for instance, for multi-model statistics, which required the model to be on a common grid and therefore has to be called after the regridding module.

10.2 Statistical preprocessors

Many preprocessors calculate statistics over data. Those preprocessors typically end with `_statistics`, e.g., *`area_statistics()`* or *`multi_model_statistics()`*. All these preprocessors support the options *operator*, which directly correspond to `iris.analysis.Aggregator` objects used to perform the statistical calculations. In addition, arbitrary keyword arguments can be passed which are directly passed to the corresponding `iris.analysis.Aggregator` object.

Note: The preprocessors *`multi_model_statistics()`* and *`ensemble_statistics()`* support the computation of multiple statistics at the same time. In these cases, they are defined by the option *statistics* (instead of *operator*), which takes a list of possible operators. Each operator can be given as single string or as dictionary. In the latter case, the dictionary needs the keyword *operator* (corresponding to the *operator* as above). All other keywords are interpreted as keyword arguments for the given operator.

Some operators support weights for some preprocessors (see following table), which are used by default. The following operators are currently fully supported; other operators might be supported too if proper keyword arguments are specified:

operator	Corresponding `Aggregator`	Weighted?[1]
`gmean`	`iris.analysis.GMEAN`	no
`hmean`	`iris.analysis.HMEAN`	no
`max`	`iris.analysis.MAX`	no
`mean`	`iris.analysis.MEAN`	yes
`median`	`iris.analysis.MEDIAN`[2]	no
`min`	`iris.analysis.MIN`	no
`peak`	`iris.analysis.PEAK`	no
`percentile`	`iris.analysis.PERCENTILE`	no
`rms`	`iris.analysis.RMS`	yes
`std_dev`	`iris.analysis.STD_DEV`	no
`sum`	`iris.analysis.SUM`	yes
`variance`	`iris.analysis.VARIANCE`	no
`wpercentile`	`iris.analysis.WPERCENTILE`	yes

[1] The following preprocessor support weighted statistics by default: *`area_statistics()`*: weighted by grid cell areas (see also *Supplementary variables (ancillary variables and cell measures)*); *`climate_statistics()`*: weighted by lengths of time intervals; *`volume_statistics()`*:

10.2.1 Examples

Calculate the global (weighted) mean:

```
preprocessors:
  global_mean:
    area_statistics:
      operator: mean
```

Calculate zonal maximum.

```
preprocessors:
  zonal_max:
    zonal_statistics:
      operator: max
```

Calculate the 95% percentile over each month separately (will result in 12 time steps, one for January, one for February, etc.):

```
preprocessors:
  monthly_percentiles:
    climate_statistics:
      period: monthly
      operator: percentile
      percent: 95.0
```

Calculate multi-model median, 5%, and 95% percentiles:

```
preprocessors:
  mm_stats:
    multi_model_statistics:
      span: overlap
      statistics:
        - operator: percentile
          percent: 5
        - operator: median
        - operator: percentile
          percent: 95
```

Calculate the global non-weighted root mean square:

```
preprocessors:
  global_mean:
    area_statistics:
      operator: rms
      weighted: false
```

Warning: The disabling of weights by specifying the keyword argument `weights: False` needs to be used with great care; from a scientific standpoint, we strongly recommend to **not** use it!

weighted by grid cell volumes (see also *Supplementary variables (ancillary variables and cell measures)*); `axis_statistics()`: weighted by corresponding coordinate bounds.

[2] `iris.analysis.MEDIAN` is not lazy, but much faster than `iris.analysis.PERCENTILE`. For a lazy median, use `percentile` with the keyword argument `percent: 50`.

10.3 Variable derivation

The variable derivation module allows to derive variables which are not in the CMIP standard data request using standard variables as input. The typical use case of this operation is the evaluation of a variable which is only available in an observational dataset but not in the models. In this case a derivation function is provided by the ESMValCore in order to calculate the variable and perform the comparison. For example, several observational datasets deliver total column ozone as observed variable (*toz*), but CMIP models only provide the ozone 3D field. In this case, a derivation function is provided to vertically integrate the ozone and obtain total column ozone for direct comparison with the observations.

To contribute a new derived variable, it is also necessary to define a name for it and to provide the corresponding CMOR table. This is to guarantee the proper metadata definition is attached to the derived data. Such custom CMOR tables are collected as part of the ESMValCore package. By default, the variable derivation will be applied only if the variable is not already available in the input data, but the derivation can be forced by setting the appropriate flag.

```
variables:
  toz:
    derive: true
    force_derivation: false
```

The required arguments for this module are two boolean switches:

- `derive`: activate variable derivation
- `force_derivation`: force variable derivation even if the variable is directly available in the input data.

See also *esmvalcore.preprocessor.derive()*. To get an overview on derivation scripts and how to implement new ones, please go to *Deriving a variable*.

10.4 CMORization and dataset-specific fixes

10.4.1 Data checking

Data preprocessed by ESMValCore is automatically checked against its CMOR definition. To reduce the impact of this check while maintaining it as reliable as possible, it is split in two parts: one will check the metadata and will be done just after loading and concatenating the data and the other one will check the data itself and will be applied after all extracting operations are applied to reduce the amount of data to process.

Checks include, but are not limited to:

- Requested coordinates are present and comply with their definition.
- Correctness of variable names, units and other metadata.
- Compliance with the valid minimum and maximum values allowed if defined.

The most relevant (i.e. a missing coordinate) will raise an error while others (i.e an incorrect long name) will be reported as a warning.

Some of those issues will be fixed automatically by the tool, including the following:

- Incorrect standard or long names.
- Incorrect units, if they can be converted to the correct ones.
- Direction of coordinates.
- Automatic clipping of longitude to 0 - 360 interval.

- Minute differences between the required and actual vertical coordinate values

10.4.2 Dataset specific fixes

Sometimes, the checker will detect errors that it can not fix by itself. ESMValCore deals with those issues by applying specific fixes for those datasets that require them. Fixes are applied at three different preprocessor steps:

- `fix_file`: apply fixes directly to a copy of the file. Copying the files is costly, so only errors that prevent Iris to load the file are fixed here. See *esmvalcore.preprocessor.fix_file()*.
- `fix_metadata`: metadata fixes are done just before concatenating the cubes loaded from different files in the final one. Automatic metadata fixes are also applied at this step. See *esmvalcore.preprocessor.fix_metadata()*.
- `fix_data`: data fixes are applied before starting any operation that will alter the data itself. Automatic data fixes are also applied at this step. See *esmvalcore.preprocessor.fix_data()*.

To get an overview on data fixes and how to implement new ones, please go to *Fixing data*.

10.5 Supplementary variables (ancillary variables and cell measures)

The following preprocessor functions either require or prefer using an ancillary variable or cell measure to perform their computations. In ESMValCore we call both types of variables "supplementary variables".

Preprocessor	Variable short name	Variable standard name
area_statistics	`areacella`, `areacello`	cell_area
mask_landsea	`sftlf`, `sftof`	land_area_fraction, sea_area_fraction
mask_landseaice	`sftgif`	land_ice_area_fraction
volume_statistics	`volcello`	ocean_volume
weighting_landsea_fraction	`sftlf`, `sftof`	land_area_fraction, sea_area_fraction

Only one of the listed variables is required. Supplementary variables can be defined in the recipe as described in *Defining supplementary variables (ancillary variables and cell measures)*. In some cases, preprocessor functions may work without supplementary variables, this is documented case by case in the preprocessor function definition. If a preprocessor function requiring supplementary variables is used without specifying these in the recipe, these will be automatically added. If the automatic selection does not give the desired result, specify the supplementary variables in the recipe as described in *Defining supplementary variables (ancillary variables and cell measures)*.

By default, supplementary variables will be removed from the variable before saving it to file because they can be as big as the main variable. To keep the supplementary variables, disable the preprocessor function *esmvalcore.preprocessor.remove_supplementary_variables()* that removes them by setting `remove_supplementary_variables: false` in the preprocessor in the recipe.

10.5.1 Examples

Compute the global mean surface air temperature, while *automatically selecting the best matching supplementary dataset*:

```
datasets:
  - dataset: BCC-ESM1
    project: CMIP6
    ensemble: r1i1p1f1
    grid: gn
  - dataset: MPI-ESM-MR
    project: CMIP5
    ensemble: r1i1p1,

preprocessors:
  global_mean:
    area_statistics:
      operator: mean

diagnostics:
  example_diagnostic:
    description: Global mean temperature.
    variables:
      tas:
        mip: Amon
        preprocessor: global_mean
        exp: historical
        timerange: '1990/2000'
        supplementary_variables:
          - short_name: areacella
            mip: fx
            exp: '*'
            activity: '*'
            ensemble: '*'
    scripts: null
```

Attach the land area fraction as an ancillary variable to surface air temperature and store both in the same file:

```
datasets:
  - dataset: BCC-ESM1
    ensemble: r1i1p1f1
    grid: gn

preprocessors:
  keep_land_area_fraction:
    remove_supplementary_variables: false

diagnostics:
  example_diagnostic:
    description: Attach land area fraction.
    variables:
      tas:
        mip: Amon
        project: CMIP6
```

(continues on next page)

(continued from previous page)

```
        preprocessor: keep_land_area_fraction
        exp: historical
        timerange: '1990/2000'
        supplementary_variables:
          - short_name: sftlf
            mip: fx
            exp: 1pctCO2
    scripts: null
```

Automatically define the required ancillary variable (`sftlf` in this case) and cell measure (`areacella`), but do not use `areacella` for dataset `BCC-ESM1`:

```
datasets:
  - dataset: BCC-ESM1
    project: CMIP6
    ensemble: r1i1p1f1
    grid: gn
    supplementary_variables:
      - short_name: areacella
        skip: true
  - dataset: MPI-ESM-MR
    project: CMIP5
    ensemble: r1i1p1

preprocessors:
  global_land_mean:
    mask_landsea:
      mask_out: sea
    area_statistics:
      operator: mean

diagnostics:
  example_diagnostic:
    description: Global mean temperature.
    variables:
      tas:
        mip: Amon
        preprocessor: global_land_mean
        exp: historical
        timerange: '1990/2000'
    scripts: null
```

10.6 Vertical interpolation

Vertical level selection is an important aspect of data preprocessing since it allows the scientist to perform a number of metrics specific to certain levels (whether it be air pressure or depth, e.g. the Quasi-Biennial-Oscillation (QBO) u30 is computed at 30 hPa). Dataset native vertical grids may not come with the desired set of levels, so an interpolation operation will be needed to regrid the data vertically. ESMValCore can perform this vertical interpolation via the `extract_levels` preprocessor. Level extraction may be done in a number of ways.

Level extraction can be done at specific values passed to `extract_levels` as `levels:` with its value a list of levels (note that the units are CMOR-standard, Pascals (Pa)):

```
preprocessors:
  preproc_select_levels_from_list:
    extract_levels:
      levels: [100000., 50000., 3000., 1000.]
      scheme: linear
```

It is also possible to extract the CMIP-specific, CMOR levels as they appear in the CMOR table, e.g. `plev10` or `plev17` or `plev19` etc:

```
preprocessors:
  preproc_select_levels_from_cmip_table:
    extract_levels:
      levels: {cmor_table: CMIP6, coordinate: plev10}
      scheme: nearest
```

Of good use is also the level extraction with values specific to a certain dataset, without the user actually polling the dataset of interest to find out the specific levels: e.g. in the example below we offer two alternatives to extract the levels and vertically regrid onto the vertical levels of `ERA-Interim`:

```
preprocessors:
  preproc_select_levels_from_dataset:
    extract_levels:
      levels: ERA-Interim
      # This also works, but allows specifying the pressure coordinate name
      # levels: {dataset: ERA-Interim, coordinate: air_pressure}
      scheme: linear_extrapolate
```

By default, vertical interpolation is performed in the dimension coordinate of the z axis. If you want to explicitly declare the z axis coordinate to use (for example, `air_pressure`' in variables that are provided in model levels and not pressure levels) you can override that automatic choice by providing the name of the desired coordinate:

```
preprocessors:
  preproc_select_levels_from_dataset:
    extract_levels:
      levels: ERA-Interim
      scheme: linear_extrapolate
      coordinate: air_pressure
```

If `coordinate` is specified, pressure levels (if present) can be converted to height levels and vice versa using the US standard atmosphere. E.g. `coordinate = altitude` will convert existing pressure levels (air_pressure) to height levels (altitude); `coordinate = air_pressure` will convert existing height levels (altitude) to pressure levels (air_pressure).

If the requested levels are very close to the values in the input data, the function will just select the available levels instead of interpolating. The meaning of 'very close' can be changed by providing the parameters:

- `rtol`
  Relative tolerance for comparing the levels in the input data to the requested levels. If the levels are sufficiently close, the requested levels will be assigned to the vertical coordinate and no interpolation will take place. The default value is 10^-7.
- `atol`
  Absolute tolerance for comparing the levels in the input data to the requested levels. If the levels are sufficiently close, the requested levels will be assigned to the vertical coordinate and no interpolation will take place. By default, *atol* will be set to 10^-7 times the mean value of of the available levels.

10.6.1 Schemes for vertical interpolation and extrapolation

The vertical interpolation currently supports the following schemes:

- `linear`: Linear interpolation without extrapolation, i.e., extrapolation points will be masked even if the source data is not a masked array.
- `linear_extrapolate`: Linear interpolation with **nearest-neighbour** extrapolation, i.e., extrapolation points will take their value from the nearest source point.
- `nearest`: Nearest-neighbour interpolation without extrapolation, i.e., extrapolation points will be masked even if the source data is not a masked array.
- `nearest_extrapolate`: Nearest-neighbour interpolation with nearest-neighbour extrapolation, i.e., extrapolation points will take their value from the nearest source point.
- See also *esmvalcore.preprocessor.extract_levels()*.
- See also `esmvalcore.preprocessor.get_cmor_levels()`.

Note: Controlling the extrapolation mode allows us to avoid situations where extrapolating values makes little physical sense (e.g. extrapolating beyond the last data point).

10.7 Weighting

10.7.1 Land/sea fraction weighting

This preprocessor allows weighting of data by land or sea fractions. In other words, this function multiplies the given input field by a fraction in the range 0-1 to account for the fact that not all grid points are completely land- or sea-covered.

The application of this preprocessor is very important for most carbon cycle variables (and other land surface outputs), which are e.g. reported in units of $kgC\ m^{-2}$. Here, the surface unit actually refers to 'square meter of land/sea' and NOT 'square meter of gridbox'. In order to integrate these globally or regionally one has to weight by both the surface quantity and the land/sea fraction.

For example, to weight an input field with the land fraction, the following preprocessor can be used:

```
preprocessors:
  preproc_weighting:
    weighting_landsea_fraction:
```

(continues on next page)

(continued from previous page)

```
      area_type: land
      exclude: ['CanESM2', 'reference_dataset']
```

Allowed arguments for the keyword `area_type` are `land` (fraction is 1 for grid cells with only land surface, 0 for grid cells with only sea surface and values in between 0 and 1 for coastal regions) and `sea` (1 for sea, 0 for land, in between for coastal regions). The optional argument `exclude` allows to exclude specific datasets from this preprocessor, which is for example useful for climate models which do not offer land/sea fraction files. This arguments also accepts the special dataset specifiers `reference_dataset` and `alternative_dataset`.

This function requires a land or sea area fraction ancillary variable. This supplementary variable, either `sftlf` or `sftof`, should be attached to the main dataset as described in *Defining supplementary variables (ancillary variables and cell measures)*.

See also *esmvalcore.preprocessor.weighting_landsea_fraction()*.

10.8 Masking

10.8.1 Introduction to masking

Certain metrics and diagnostics need to be computed and performed on specific domains on the globe. The preprocessor supports filtering the input data on continents, oceans/seas and ice. This is achieved by masking the model data and keeping only the values associated with grid points that correspond to, e.g., land, ocean or ice surfaces, as specified by the user. Where possible, the masking is realized using the standard mask files provided together with the model data as part of the CMIP data request (the so-called ancillary variable). In the absence of these files, the Natural Earth masks are used: although these are not model-specific, they represent a good approximation since they have a much higher resolution than most of the models and they are regularly updated with changing geographical features.

10.8.2 Land-sea masking

To mask out a certain domain (e.g., sea) in the preprocessor, `mask_landsea` can be used:

```
preprocessors:
  preproc_mask:
    mask_landsea:
      mask_out: sea
```

and requires only one argument: `mask_out`: either `land` or `sea`.

This function prefers using a land or sea area fraction ancillary variable, but if it is not available it will compute a mask based on Natural Earth shapefiles. This supplementary variable, either `sftlf` or `sftof`, can be attached to the main dataset as described in *Defining supplementary variables (ancillary variables and cell measures)*.

If the corresponding ancillary variable is not available (which is the case for some models and almost all observational datasets), the preprocessor attempts to mask the data using Natural Earth mask files (that are vectorized rasters). As mentioned above, the spatial resolution of the the Natural Earth masks are much higher than any typical global model (10m for land and glaciated areas and 50m for ocean masks).

See also *esmvalcore.preprocessor.mask_landsea()*.

10.8.3 Ice masking

For masking out ice sheets, the preprocessor uses a different function, to ensure that both land and sea or ice can be masked out without losing generality. To mask ice out, `mask_landseaice` can be used:

```
preprocessors:
  preproc_mask:
    mask_landseaice:
      mask_out: ice
```

and requires only one argument: `mask_out`: either `landsea` or `ice`.

This function requires a land ice area fraction ancillary variable. This supplementary variable `sftgif` should be attached to the main dataset as described in *Defining supplementary variables (ancillary variables and cell measures)*.

See also *esmvalcore.preprocessor.mask_landseaice()*.

10.8.4 Glaciated masking

For masking out glaciated areas a Natural Earth shapefile is used. To mask glaciated areas out, `mask_glaciated` can be used:

```
preprocessors:
  preproc_mask:
    mask_glaciated:
      mask_out: glaciated
```

and it requires only one argument: `mask_out`: only `glaciated`.

See also *esmvalcore.preprocessor.mask_landseaice()*.

10.8.5 Missing values masks

Missing (masked) values can be a nuisance especially when dealing with multi-model ensembles and having to compute multi-model statistics; different numbers of missing data from dataset to dataset may introduce biases and artificially assign more weight to the datasets that have less missing data. This is handled via the missing values masks: two types of such masks are available, one for the multi-model case and another for the single model case.

The multi-model missing values mask (`mask_fillvalues`) is a preprocessor step that usually comes after all the single-model steps (regridding, area selection etc) have been performed; in a nutshell, it combines missing values masks from individual models into a multi-model missing values mask; the individual model masks are built according to common criteria: the user chooses a time window in which missing data points are counted, and if the number of missing data points relative to the number of total data points in a window is less than a chosen fractional threshold, the window is discarded i.e. all the points in the window are masked (set to missing).

```
preprocessors:
  missing_values_preprocessor:
    mask_fillvalues:
      threshold_fraction: 0.95
      min_value: 19.0
      time_window: 10.0
```

In the example above, the fractional threshold for missing data vs. total data is set to 95% and the time window is set to 10.0 (units of the time coordinate units). Optionally, a minimum value threshold can be applied, in this case it is set to 19.0 (in units of the variable units).

See also *esmvalcore.preprocessor.mask_fillvalues()*.

10.8.6 Common mask for multiple models

To create a combined multi-model mask (all the masks from all the analyzed datasets combined into a single mask using a logical OR), the preprocessor `mask_multimodel` can be used. In contrast to `mask_fillvalues`, `mask_multimodel` does not expect that the datasets have a `time` coordinate, but works on datasets with arbitrary (but identical) coordinates. After `mask_multimodel`, all involved datasets have an identical mask.

See also *esmvalcore.preprocessor.mask_multimodel()*.

10.8.7 Minimum, maximum and interval masking

Thresholding on minimum and maximum accepted data values can also be performed: masks are constructed based on the results of thresholding; inside and outside interval thresholding and masking can also be performed. These functions are `mask_above_threshold`, `mask_below_threshold`, `mask_inside_range`, and `mask_outside_range`.

These functions always take a cube as first argument and either `threshold` for threshold masking or the pair `minimum`, `maximum` for interval masking.

See also *esmvalcore.preprocessor.mask_above_threshold()* and related functions.

10.9 Horizontal regridding

Regridding is necessary when various datasets are available on a variety of *lat-lon* grids and they need to be brought together on a common grid (for various statistical operations e.g. multi-model statistics or for e.g. direct inter-comparison or comparison with observational datasets). Regridding is conceptually a very similar process to interpolation (in fact, the regridder engine uses interpolation and extrapolation, with various schemes). The primary difference is that interpolation is based on sample data points, while regridding is based on the horizontal grid of another cube (the reference grid). If the horizontal grids of a cube and its reference grid are sufficiently the same, regridding is automatically and silently skipped for performance reasons.

The underlying regridding mechanism in ESMValCore uses `iris.cube.Cube.regrid` from Iris.

The use of the horizontal regridding functionality is flexible depending on what type of reference grid and what interpolation scheme is preferred. Below we show a few examples.

10.9.1 Regridding on a reference dataset grid

The example below shows how to regrid on the reference dataset `ERA-Interim` (observational data, but just as well CMIP, obs4MIPs, or ana4mips datasets can be used); in this case the *scheme* is *linear*.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: ERA-Interim
      scheme: linear
```

10.9.2 Regridding on an `MxN` grid specification

The example below shows how to regrid on a reference grid with a cell specification of `2.5x2.5` degrees. This is similar to regridding on reference datasets, but in the previous case the reference dataset grid cell specifications are not necessarily known a priori. Regridding on an `MxN` cell specification is oftentimes used when operating on localized data.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      scheme: nearest
```

In this case the `NearestNeighbour` interpolation scheme is used (see below for scheme definitions).

When using a `MxN` type of grid it is possible to offset the grid cell centrepoints using the *lat_offset* and `lon_offset` arguments:

- `lat_offset`: offsets the grid centers of the latitude coordinate w.r.t. the pole by half a grid step;
- `lon_offset`: offsets the grid centers of the longitude coordinate w.r.t. Greenwich meridian by half a grid step.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      lon_offset: True
      lat_offset: True
      scheme: nearest
```

10.9.3 Regridding to a regional target grid specification

This example shows how to regrid to a regional target grid specification. This is useful if both a `regrid` and `extract_region` step are necessary.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid:
        start_longitude: 40
        end_longitude: 60
        step_longitude: 2
        start_latitude: -10
        end_latitude: 30
        step_latitude: 2
      scheme: nearest
```

This defines a grid ranging from 40° to 60° longitude with 2° steps, and -10° to 30° latitude with 2° steps. If `end_longitude` or `end_latitude` do not fall on the grid (e.g., `end_longitude: 61`), it cuts off at the nearest previous value (e.g. `60`).

The longitude coordinates will wrap around the globe if necessary, i.e. `start_longitude: 350`, `end_longitude: 370` is valid input.

The arguments are defined below:

- `start_latitude`: Latitude value of the first grid cell center (start point). The grid includes this value.
- `end_latitude`: Latitude value of the last grid cell center (end point). The grid includes this value only if it falls on a grid point. Otherwise, it cuts off at the previous value.
- `step_latitude`: Latitude distance between the centers of two neighbouring cells.
- `start_longitude`: Latitude value of the first grid cell center (start point). The grid includes this value.
- `end_longitude`: Longitude value of the last grid cell center (end point). The grid includes this value only if it falls on a grid point. Otherwise, it cuts off at the previous value.
- `step_longitude`: Longitude distance between the centers of two neighbouring cells.

10.9.4 Regridding (interpolation, extrapolation) schemes

ESMValCore has a number of built-in regridding schemes, which are presented in *Built-in regridding schemes*. Additionally, it is also possible to use third party regridding schemes designed for use with Iris. This is explained in *Generic regridding schemes*.

Built-in regridding schemes

The schemes used for the interpolation and extrapolation operations needed by the horizontal regridding functionality directly map to their corresponding implementations in `iris`:

- `linear`: Linear interpolation without extrapolation, i.e., extrapolation points will be masked even if the source data is not a masked array (uses `Linear(extrapolation_mode='mask')`, see `iris.analysis.Linear`).
- `linear_extrapolate`: Linear interpolation with extrapolation, i.e., extrapolation points will be calculated by extending the gradient of the closest two points (uses `Linear(extrapolation_mode='extrapolate')`, see `iris.analysis.Linear`).
- `nearest`: Nearest-neighbour interpolation without extrapolation, i.e., extrapolation points will be masked even if the source data is not a masked array (uses `Nearest(extrapolation_mode='mask')`, see `iris.analysis.Nearest`).
- `area_weighted`: Area-weighted regridding (uses `AreaWeighted()`, see `iris.analysis.AreaWeighted`).
- `unstructured_nearest`: Nearest-neighbour interpolation for unstructured grids (uses `UnstructuredNearest()`, see `iris.analysis.UnstructuredNearest`).

See also *esmvalcore.preprocessor.regrid()*

Note: Controlling the extrapolation mode allows us to avoid situations where extrapolating values makes little physical sense (e.g. extrapolating beyond the last data point).

Note: The regridding mechanism is (at the moment) done with fully realized data in memory, so depending on how fine the target grid is, it may use a rather large amount of memory. Empirically target grids of up to `0.5x0.5` degrees should not produce any memory-related issues, but be advised that for resolutions of `< 0.5` degrees the regridding becomes very slow and will use a lot of memory.

Generic regridding schemes

Iris' regridding is based around the flexible use of so-called regridding schemes. These are classes that know how to transform a source cube with a given grid into the grid defined by a given target cube. Iris itself provides a number of useful schemes, but they are largely limited to work with simple, regular grids. Other schemes can be provided independently. This is interesting when special regridding-needs arise or when more involved grids and meshes need to be considered. Furthermore, it may be desirable to have finer control over the parameters of the scheme than is afforded by the built-in schemes described above.

To facilitate this, the *regrid()* preprocessor allows the use of any scheme designed for Iris. The scheme must be installed and importable. To use this feature, the `scheme` key passed to the preprocessor must be a dictionary instead of a simple string that contains all necessary information. That includes a `reference` to the desired scheme itself, as well as any arguments that should be passed through to the scheme. For example, the following shows the use of the built-in scheme `iris.analysis.AreaWeighted` with a custom threshold for missing data tolerance.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      scheme:
        reference: iris.analysis:AreaWeighted
        mdtol: 0.7
```

The value of the `reference` key has two parts that are separated by a `:` with no surrounding spaces. The first part is an importable Python module, the second refers to the scheme, i.e. some callable that will be called with the remaining entries of the `scheme` dictionary passed as keyword arguments.

One package that aims to capitalize on the support for unstructured meshes introduced in Iris 3.2 is iris-esmf-regrid. It aims to provide lazy regridding for structured regular and irregular grids, as well as unstructured meshes. An example of its usage in a preprocessor is:

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      scheme:
        reference: esmf_regrid.schemes:ESMFAreaWeighted
        mdtol: 0.7
```

Additionally, the use of generic schemes that take source and target grid cubes as arguments is also supported. The call function for such schemes must be defined as *(src_cube, grid_cube, **kwargs)* and they must return *iris.cube.Cube* objects. The *regrid* module will automatically pass the source and grid cubes as inputs of the scheme. An example of this usage is the `regrid_rectilinear_to_rectilinear()` scheme available in iris-esmf-regrid:

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      scheme:
        reference: esmf_regrid.schemes:regrid_rectilinear_to_rectilinear
        mdtol: 0.7
```

10.10 Ensemble statistics

For certain use cases it may be desirable to compute ensemble statistics. For example to prevent models with many ensemble members getting excessive weight in the multi-model statistics functions.

Theoretically, ensemble statistics are a special case (grouped) multi-model statistics. This grouping is performed taking into account the dataset tags *project*, *dataset*, *experiment*, and (if present) *sub_experiment*. However, they should typically be computed earlier in the workflow. Moreover, because multiple ensemble members of the same model are typically more consistent/homogeneous than datasets from different models, the implementation is more straightforward and can benefit from lazy evaluation and more efficient computation.

The preprocessor takes a list of statistics as input:

```
preprocessors:
  example_preprocessor:
    ensemble_statistics:
      statistics: [mean, median]
```

Additional keyword arguments can be given by using a dictionary:

```
preprocessors:
  example_preprocessor:
    ensemble_statistics:
      statistics:
        - operator: percentile
          percent: 20
        - operator: median
```

This preprocessor function exposes the iris analysis package, and works with all (capitalized) statistics from the `iris.analysis` package that can be executed without additional arguments. See *Statistical preprocessors* for more details on supported statistics.

Note that `ensemble_statistics` will not return the single model and ensemble files, only the requested ensemble statistics results.

In case of wanting to save both individual ensemble members as well as the statistic results, the preprocessor chains could be defined as:

```
preprocessors:
  everything_else: &everything_else
    area_statistics: ...
    regrid_time: ...
  multimodel:
    <<: *everything_else
    ensemble_statistics:

variables:
  tas_datasets:
    short_name: tas
    preprocessor: everything_else
    ...
  tas_multimodel:
    short_name: tas
    preprocessor: multimodel
    ...
```

See also *esmvalcore.preprocessor.ensemble_statistics()*.

10.11 Multi-model statistics

Computing multi-model statistics is an integral part of model analysis and evaluation: individual models display a variety of biases depending on model set-up, initial conditions, forcings and implementation; comparing model data to observational data, these biases have a significantly lower statistical impact when using a multi-model ensemble. ESMValCore has the capability of computing a number of multi-model statistical measures: using the preprocessor module `multi_model_statistics` will enable the user for example to ask for either a multi-model `mean`, `median`, `max`, `min`, `std_dev`, and / or `percentile` with a set of argument parameters passed to `multi_model_statistics`. See *Statistical preprocessors* for more details on supported statistics. Percentiles can be specified with additional keyword arguments using the syntax `statistics: [{operator: percentile, percent: xx}]`.

Restrictive computation is also available by excluding any set of models that the user will not want to include in the statistics (by setting `exclude: [excluded models list]` argument).

Input datasets may have different time coordinates. Apart from that, all dimensions must match. Statistics can be computed across overlapping times only (`span: overlap`) or across the full time span of the combined models (`span: full`). The preprocessor sets a common time coordinate on all datasets. As the number of days in a year may vary between calendars, (sub-)daily data with different calendars are not supported. The preprocessor saves both the input single model files as well as the multi-model results. In case you do not want to keep the single model files, set the parameter `keep_input_datasets` to `false` (default value is `true`). To remove scalar coordinates before merging input datasets into the multi-dataset cube, use the option `ignore_scalar_coords: true`. The resulting multi-dataset cube will not have scalar coordinates in this case. This ensures that differences in scalar coordinates in the input datasets are ignored, which is helpful if you encounter a `ValueError: Multi-model statistics failed to merge input cubes into a single array` with `Coordinates in cube.aux_coords (scalar) differ`. Some special scalar coordinates which are expected to differ across cubes (*p0* and *ptop*) are always removed.

```yaml
preprocessors:
  multi_model_save_input:
    multi_model_statistics:
      span: overlap
      statistics: [mean, median]
      exclude: [NCEP-NCAR-R1]
  multi_model_without_saving_input:
    multi_model_statistics:
      span: overlap
      statistics: [mean, median]
      exclude: [NCEP-NCAR-R1]
      keep_input_datasets: false
      ignore_scalar_coords: true
  multi_model_percentiles_5_95:
    multi_model_statistics:
      span: overlap
      statistics:
        - operator: percentile
          percent: 5
        - operator: percentile
          percent: 95
```

Multi-model statistics also supports a `groupby` argument. You can group by any dataset key (`project`, `experiment`, etc.) or a combination of keys in a list. You can also add an arbitrary tag to a dataset definition and then group by that tag. When using this preprocessor in conjunction with *ensemble statistics* preprocessor, you can group by `ensemble_statistics` as well. For example:

```
datasets:
  - {dataset: CanESM2, exp: historical, ensemble: "r(1:2)i1p1"}
  - {dataset: CCSM4, exp: historical, ensemble: "r(1:2)i1p1"}

preprocessors:
  example_preprocessor:
    ensemble_statistics:
      statistics: [median, mean]
    multi_model_statistics:
      span: overlap
      statistics: [min, max]
      groupby: [ensemble_statistics]
      exclude: [NCEP-NCAR-R1]
```

This will first compute ensemble mean and median, and then compute the multi-model min and max separately for the ensemble means and medians. Note that this combination will not save the individual ensemble members, only the ensemble and multimodel statistics results.

When grouping by a tag not defined in all datasets, the datasets missing the tag will be grouped together. In the example below, datasets *UKESM* and *ERA5* would belong to the same group, while the other datasets would belong to either `group1` or `group2`

```
datasets:
  - {dataset: CanESM2, exp: historical, ensemble: "r(1:2)i1p1", tag: 'group1'}
  - {dataset: CanESM5, exp: historical, ensemble: "r(1:2)i1p1", tag: 'group2'}
  - {dataset: CCSM4, exp: historical, ensemble: "r(1:2)i1p1", tag: 'group2'}
  - {dataset: UKESM, exp: historical, ensemble: "r(1:2)i1p1"}
  - {dataset: ERA5}

preprocessors:
  example_preprocessor:
    multi_model_statistics:
      span: overlap
      statistics: [min, max]
      groupby: [tag]
```

Note that those datasets can be excluded if listed in the `exclude` option.

See also *esmvalcore.preprocessor.multi_model_statistics()*.

10.12 Time manipulation

The `_time.py` module contains the following preprocessor functions:

- *extract_time*: Extract a time range from a cube.
- *extract_season*: Extract only the times that occur within a specific season.
- *extract_month*: Extract only the times that occur within a specific month.
- *hourly_statistics*: Compute intra-day statistics
- *daily_statistics*: Compute statistics for each day
- *monthly_statistics*: Compute statistics for each month

- *seasonal_statistics*: Compute statistics for each season
- *annual_statistics*: Compute statistics for each year
- *decadal_statistics*: Compute statistics for each decade
- *climate_statistics*: Compute statistics for the full period
- *resample_time*: Resample data
- *resample_hours*: Convert between N-hourly frequencies by resampling
- *anomalies*: Compute (standardized) anomalies
- *regrid_time*: Aligns the time axis of each dataset to have common time points and calendars.
- *timeseries_filter*: Allows application of a filter to the time-series data.

Statistics functions are applied by default in the order they appear in the list. For example, the following example applied to hourly data will retrieve the minimum values for the full period (by season) of the monthly mean of the daily maximum of any given variable.

```
daily_statistics:
  operator: max

monthly_statistics:
  operator: mean

climate_statistics:
  operator: min
  period: season
```

10.12.1 extract_time

This function subsets a dataset between two points in times. It removes all times in the dataset before the first time and after the last time point. The required arguments are relatively self explanatory:

- start_year
- start_month
- start_day
- end_year
- end_month
- end_day

These start and end points are set using the datasets native calendar. All six arguments should be given as integers - the named month string will not be accepted.

See also *esmvalcore.preprocessor.extract_time()*.

10.12.2 extract_season

Extract only the times that occur within a specific season.

This function only has one argument: `season`. This is the named season to extract, i.e. DJF, MAM, JJA, SON, but also all other sequentially correct combinations, e.g. JJAS.

Note that this function does not change the time resolution. If your original data is in monthly time resolution, then this function will return three monthly datapoints per year.

If you want the seasonal average, then this function needs to be combined with the seasonal_mean function, below.

See also *esmvalcore.preprocessor.extract_season()*.

10.12.3 extract_month

The function extracts the times that occur within a specific month. This function only has one argument: `month`. This value should be an integer between 1 and 12 as the named month string will not be accepted.

See also *esmvalcore.preprocessor.extract_month()*.

10.12.4 hourly_statistics

This function produces statistics at a x-hourly frequency.

Parameters:

- *hour*: Number of hours per period. Must be a divisor of 24, i.e., (1, 2, 3, 4, 6, 8, 12).
- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also *esmvalcore.preprocessor.hourly_statistics()*.

10.12.5 daily_statistics

This function produces statistics for each day in the dataset.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also *esmvalcore.preprocessor.daily_statistics()*.

10.12.6 monthly_statistics

This function produces statistics for each month in the dataset.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also *esmvalcore.preprocessor.monthly_statistics()*.

10.12.7 seasonal_statistics

This function produces statistics for each season (default: `[DJF, MAM, JJA, SON]` or custom seasons e.g. `[JJAS, ONDJFMAM]`) in the dataset. Note that this function will not check for missing time points. For instance, if you are looking at the DJF field, but your datasets starts on January 1st, the first DJF field will only contain data from January and February.

We recommend using the extract_time to start the dataset from the following December and remove such biased initial data points.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- *seasons*: Seasons to build statistics. Default is `'[DJF, MAM, JJA, SON]'`.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also *esmvalcore.preprocessor.seasonal_statistics()*.

10.12.8 annual_statistics

This function produces statistics for each year.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also *esmvalcore.preprocessor.annual_statistics()*.

10.12.9 decadal_statistics

This function produces statistics for each decade.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also *esmvalcore.preprocessor.decadal_statistics()*.

10.12.10 climate_statistics

This function produces statistics for the whole dataset. It can produce scalars (if the full period is chosen) or hourly, daily, monthly or seasonal statistics.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics. Default is *mean*.
- *period*: Define the granularity of the statistics: get values for the full period, for each month, day of year or hour of day. Available periods: *full*, *season*, *seasonal*, *monthly*, *month*, *mon*, *daily*, *day*, *hourly*, *hour*, *hr*. Default is *full*.
- *seasons*: if period 'seasonal' or 'season' allows to set custom seasons. Default is `'[DJF, MAM, JJA, SON]'`.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

Note: Some operations are weighted by the time coordinate by default, i.e., the length of the time intervals. See *Statistical preprocessors* for more details on supported statistics. For *sum*, the units of the resulting cube are multiplied by the corresponding time units (e.g., days).

Examples:

- Monthly climatology:

```
climate_statistics:
  operator: mean
  period: month
```

- Daily maximum for the full period:

```
climate_statistics:
  operator: max
  period: day
```

- Minimum value in the period:

```
climate_statistics:
  operator: min
  period: full
```

- 80% percentile for each month:

```
climate_statistics:
  period: month
  operator: percentile
  percent: 80
```

See also *esmvalcore.preprocessor.climate_statistics()*.

10.12.11 resample_time

This function changes the frequency of the data in the cube by extracting the timesteps that meet the criteria. It is important to note that it is mainly meant to be used with instantaneous data.

Parameters:

- month: Extract only timesteps from the given month or do nothing if None. Default is *None*
- day: Extract only timesteps from the given day of month or do nothing if None. Default is *None*
- hour: Extract only timesteps from the given hour or do nothing if None. Default is *None*

Examples:

- Hourly data to daily:

```
resample_time:
  hour: 12
```

- Hourly data to monthly:

```
resample_time:
  hour: 12
  day: 15
```

- Daily data to monthly:

```
resample_time:
  day: 15
```

See also *esmvalcore.preprocessor.resample_time()*.

resample_hours:

10.12.12 resample_hours

This function changes the frequency of the data in the cube by extracting the timesteps that belongs to the desired frequency. It is important to note that it is mainly mean to be used with instantaneous data

Parameters:

- interval: New frequency of the data. Must be a divisor of 24
- offset: First desired hour. Default 0. Must be lower than the interval

Examples:

- Convert to 12-hourly, by getting timesteps at 0:00 and 12:00:

```
resample_hours:
  hours: 12
```

- Convert to 12-hourly, by getting timesteps at 6:00 and 18:00:

```
resample_hours:
  hours: 12
  offset: 6
```

See also *esmvalcore.preprocessor.resample_hours()*.

10.12.13 anomalies

This function computes the anomalies for the whole dataset. It can compute anomalies from the full, seasonal, monthly, daily and hourly climatologies. Optionally standardized anomalies can be calculated.

Parameters:

- period: define the granularity of the climatology to use: full period, seasonal, monthly, daily or hourly. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day', 'hourly', 'hour', 'hr'. Default is 'full'
- reference: Time slice to use as the reference to compute the climatology on. Can be 'null' to use the full cube or a dictionary with the parameters from *extract_time*. Default is null
- standardize: if true calculate standardized anomalies (default: false)
- seasons: if period 'seasonal' or 'season' allows to set custom seasons. Default is '[DJF, MAM, JJA, SON]'

Examples:

- Anomalies from the full period climatology:

```
anomalies:
```

- Anomalies from the full period monthly climatology:

```
anomalies:
  period: month
```

- Standardized anomalies from the full period climatology:

```
anomalies:
  standardized: true
```

- Standardized Anomalies from the 1979-2000 monthly climatology:

```
anomalies:
  period: month
  reference:
    start_year: 1979
    start_month: 1
    start_day: 1
    end_year: 2000
    end_month: 12
    end_day: 31
  standardize: true
```

See also *esmvalcore.preprocessor.anomalies()*.

10.12.14 regrid_time

This function aligns the time points of each component dataset so that the Iris cubes from different datasets can be subtracted. The operation makes the datasets time points common; it also resets the time bounds and auxiliary coordinates to reflect the artificially shifted time points. Current implementation for monthly and daily data; the `frequency` is set automatically from the variable CMOR table unless a custom `frequency` is set manually by the user in recipe.

See also *esmvalcore.preprocessor.regrid_time()*.

10.12.15 timeseries_filter

This function allows the user to apply a filter to the timeseries data. This filter may be of the user's choice (currently only the `low-pass` Lanczos filter is implemented); the implementation is inspired by this iris example and uses aggregation via `iris.cube.Cube.rolling_window`.

Parameters:

- window: the length of the filter window (in units of cube time coordinate).
- span: period (number of months/days, depending on data frequency) on which weights should be computed e.g. for 2-yearly: span = 24 (2 x 12 months). Make sure span has the same units as the data cube time coordinate.
- filter_type: the type of filter to be applied; default 'lowpass'. Available types: 'lowpass'.
- filter_stats: the type of statistic to aggregate on the rolling window; default 'sum'. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'.

Examples:

- Lowpass filter with a monthly mean as operator:

```
timeseries_filter:
    window: 3  # 3-monthly filter window
    span: 12   # weights computed on the first year
    filter_type: lowpass  # low-pass filter
    filter_stats: mean    # 3-monthly mean lowpass filter
```

See also *esmvalcore.preprocessor.timeseries_filter()*.

10.13 Area manipulation

The area manipulation module contains the following preprocessor functions:

- *extract_coordinate_points*: Extract a point with arbitrary coordinates given an interpolation scheme.
- *extract_region*: Extract a region from a cube based on `lat/lon` corners.
- *extract_named_regions*: Extract a specific region from in the region coordinate.
- *extract_shape*: Extract a region defined by a shapefile.
- *extract_point*: Extract a single point (with interpolation)
- *extract_location*: Extract a single point by its location (with interpolation)

- *zonal_statistics*: Compute zonal statistics.
- *meridional_statistics*: Compute meridional statistics.
- *area_statistics*: Compute area statistics.

10.13.1 extract_coordinate_points

This function extracts points with given coordinates, following either a `linear` or a `nearest` interpolation scheme. The resulting point cube will match the respective coordinates to those of the input coordinates. If the input coordinate is a scalar, the dimension will be a scalar in the output cube.

If the point to be extracted has at least one of the coordinate point values outside the interval of the cube's same coordinate values, then no extrapolation will be performed, and the resulting extracted cube will have fully masked data.

Examples:

- Extract a point from coordinate *grid_latitude* with given coordinate value 26.0:

```
extract_coordinate_points:
  definition:
    grid_latitude: 26.
  scheme: nearest
```

See also *esmvalcore.preprocessor.extract_coordinate_points()*.

10.13.2 extract_region

This function returns a subset of the data on the rectangular region requested. The boundaries of the region are provided as latitude and longitude coordinates in the arguments:

- `start_longitude`
- `end_longitude`
- `start_latitude`
- `end_latitude`

Note that this function can only be used to extract a rectangular region. Use `extract_shape` to extract any other shaped region from a shapefile.

If the grid is irregular, the returned region retains the original coordinates, but is cropped to a rectangular bounding box defined by the start/end coordinates. The deselected area inside the region is masked.

See also *esmvalcore.preprocessor.extract_region()*.

10.13.3 extract_named_regions

This function extracts a specific named region from the data. This function takes the following argument: `regions` which is either a string or a list of strings of named regions. Note that the dataset must have a `region` coordinate which includes a list of strings as values. This function then matches the named regions against the requested string.

See also *esmvalcore.preprocessor.extract_named_regions()*.

10.13.4 extract_shape

Extract a shape or a representative point for this shape from the data.

Parameters:

- `shapefile`: path to the shapefile containing the geometry of the region to be extracted. If the file contains multiple shapes behaviour depends on the `decomposed` parameter. This path can be relative to `auxiliary_data_dir` defined in the *User configuration file* or relative to `esmvalcore/preprocessor/shapefiles` (in that priority order). Alternatively, a string (see "Shapefile name" below) can be given to load one of the following shapefiles that are shipped with ESMValCore:

Shapefile name	Description	Reference
ar6	IPCC WG1 reference regions (v4) used in Assessment Report 6	https://doi.org/10.5281/zenodo.5176260

- `method`: the method to select the region, selecting either all points contained by the shape or a single representative point. Choose either *'contains'* or *'representative'*. If not a single grid point is contained in the shape, a representative point will be selected.
- `crop`: by default *extract_region* will be used to crop the data to a minimal rectangular region containing the shape. Set to `false` to only mask data outside the shape. Data on irregular grids will not be cropped.
- `decomposed`: by default `false`; in this case the union of all the regions in the shapefile is masked out. If set to `true`, the regions in the shapefiles are masked out separately and the output cube will have an additional dimension `shape_id` describing the requested regions.
- `ids`: Shapes to be read from the shapefile. Can be given as:
  - `list`: IDs are assigned from the attributes `name`, `NAME`, `Name`, `id`, or `ID` (in that priority order; the first one available is used). If none of these attributes are available in the shapefile, assume that the given *ids* correspond to the reading order of the individual shapes. So, for example, if a file has both `name` and `id` attributes, the ids will be assigned from `name`. If the file only has the `id` attribute, it will be taken from it and if no `name` nor `id` attributes are present, an integer ID starting from 0 will be assigned automatically when reading the shapes. We discourage to rely on this last behaviour as we can not assure that the reading order will be the same on different platforms, so we encourage you to specify a custom attribute using a `dict` (see below) instead. Note: An empty list is interpreted as `ids=None` (see below).
  - `dict`: IDs (dictionary value; `list` of `str`) are assigned from attribute given as dictionary key (`str`). Only dictionaries with length 1 are supported. Example: `ids={'Acronym': ['GIC', 'WNA']}`.
  - *None*: select all available regions from the shapefile.

Examples:

- Extract the shape of the river Elbe from a shapefile:

```
extract_shape:
  shapefile: Elbe.shp
  method: contains
```

- Extract the shape of several countries:

```
extract_shape:
shapefile: NaturalEarth/Countries/ne_110m_admin_0_countries.shp
decomposed: True
method: contains
ids:
  - Spain
  - France
  - Italy
  - United Kingdom
  - Taiwan
```

- Extract European AR6 regions:

```
extract_shape:
  shapefile: ar6
  method: contains
  ids:
    Acronym:
      - NEU
      - WCE
      - MED
```

See also *esmvalcore.preprocessor.extract_shape()*.

10.13.5 extract_point

Extract a single point from the data. This is done using either nearest or linear interpolation.

Returns a cube with the extracted point(s), and with adjusted latitude and longitude coordinates (see below).

Multiple points can also be extracted, by supplying an array of latitude and/or longitude coordinates. The resulting point cube will match the respective latitude and longitude coordinate to those of the input coordinates. If the input coordinate is a scalar, the dimension will be missing in the output cube (that is, it will be a scalar).

If the point to be extracted has at least one of the coordinate point values outside the interval of the cube's same coordinate values, then no extrapolation will be performed, and the resulting extracted cube will have fully masked data.

Parameters:

- `cube`: the input dataset cube.
- `latitude`, `longitude`: coordinates (as floating point values) of the point to be extracted. Either (or both) can also be an array of floating point values.
- `scheme`: interpolation scheme: either `'linear'` or `'nearest'`. There is no default.

See also *esmvalcore.preprocessor.extract_point()*.

10.13.6 extract_location

Extract a single point using a location name, with interpolation (either linear or nearest). This preprocessor extracts a single location point from a cube, according to the given interpolation scheme `scheme`. The function retrieves the coordinates of the location and then calls the *esmvalcore.preprocessor.extract_point()* preprocessor. It can be used to locate cities and villages, but also mountains or other geographical locations.

Note: Note that this function's geolocator application needs a working internet connection.

Parameters:

- *cube*: the input dataset cube to extract a point from.
- *location*: the reference location. Examples: 'mount everest', 'romania', 'new york, usa'. Raises ValueError if none supplied.
- *scheme* : interpolation scheme. *linear* or *nearest*. There is no default, raises ValueError if none supplied.

See also *esmvalcore.preprocessor.extract_location()*.

10.13.7 zonal_statistics

The function calculates the zonal statistics by applying an operator along the longitude coordinate.

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also `esmvalcore.preprocessor.zonal_means()`.

10.13.8 meridional_statistics

The function calculates the meridional statistics by applying an operator along the latitude coordinate. This function takes one argument:

Parameters:

- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

See also `esmvalcore.preprocessor.meridional_means()`.

10.13.9 area_statistics

This function calculates statistics over a region. It takes one argument, `operator`, which is the name of the operation to apply.

This function can be used to apply several different operations in the horizontal plane: for example, mean, sum, standard deviation, median, variance, minimum, maximum and root mean square. Some operations are grid cell area weighted by default. For sums, the units of the resulting cubes are multiplied by m ^{2}. See *Statistical preprocessors* for more details on supported statistics.

Note that this function is applied over the entire dataset. If only a specific region, depth layer or time period is required, then those regions need to be removed using other preprocessor operations in advance.

For weighted statistics, this function requires a cell area cell measure, unless the coordinates of the input data are regular 1D latitude and longitude coordinates so the cell areas can be computed internally. The required supplementary variable, either `areacella` for atmospheric variables or `areacello` for ocean variables, can be attached to the main dataset as described in *Defining supplementary variables (ancillary variables and cell measures)*.

See also *esmvalcore.preprocessor.area_statistics()*.

10.14 Volume manipulation

The `_volume.py` module contains the following preprocessor functions:

- `axis_statistics`: Perform operations along a given axis.
- `extract_volume`: Extract a specific depth range from a cube.
- `volume_statistics`: Calculate the volume-weighted average.
- `depth_integration`: Integrate over the depth dimension.
- `extract_transect`: Extract data along a line of constant latitude or longitude.
- `extract_trajectory`: Extract data along a specified trajectory.

10.14.1 extract_volume

Extract a specific range in the z-direction from a cube. The range is given as an interval that can be:

- open `(z_min, z_max)`, in which the extracted range does not include `z_min` nor `z_max`.
- closed `[z_min, z_max]`, in which the extracted includes both `z_min` and `z_max`.
- left closed `[z_min, z_max)`, in which the extracted range includes `z_min` but not `z_max`.
- right closed `(z_min, z_max]`, in which the extracted range includes `z_max` but not `z_min`.

The extraction is performed by applying a constraint on the coordinate values, without any kind of interpolation.

This function takes four arguments:

- `z_min` to define the minimum value of the range to extract in the z-direction.
- `z_max` to define the maximum value of the range to extract in the z-direction.
- **`interval_bounds` to define whether the bounds of the interval are `open`, `closed`,** `left_closed` or `right_closed`. Default is `open`.
- **`nearest_value` to extract a range taking into account the values of the z-coordinate that** are closest to `z_min` and `z_max`. Default is `False`.

As the coordinate points are likely to vary depending on the dataset, sometimes it might be useful to adjust the given `z_min` and `z_max` values to the values of the coordinate points before performing an extraction. This behaviour can be achieved by setting the `nearest_value` argument to `True`.

For example, in a cube with `z_coord = [0., 1.5, 2.6., 3.8., 5.4]`, the preprocessor below:

```
preprocessors:
  extract_volume:
    z_min: 1.
    z_max: 5.
    interval_bounds: 'closed'
```

would return a cube with a `z_coord` defined as `z_coord = [1.5, 2.6., 3.8.]`, since these are the values that strictly fall into the range given by `[z_min=1, z_max=5]`.

Whereas setting `earest_value: True`:

```
preprocessors:
  extract_volume:
    z_min: 1.
    z_max: 5.
    interval_bounds: 'closed'
    nearest_value: True
```

would return a cube with a `z_coord` defined as `z_coord = [1.5, 2.6., 3.8., 5.4]`, since `z_max = 5` is closest to the coordinate point `z = 5.4` than it is to `z = 3.8`.

Note that this preprocessor requires the requested z-coordinate range to be the same sign as the Iris cube. That is, if the cube has z-coordinate as negative, then `z_min` and `z_max` need to be negative numbers.

See also *esmvalcore.preprocessor.extract_volume()*.

10.14.2 volume_statistics

This function calculates the volume-weighted average across three dimensions, but maintains the time dimension.

This function takes the argument: *operator*, which defines the operation to apply over the volume. At the moment, only *mean* is supported. By default, the *mean* operation is weighted by the grid cell volumes.

For weighted statistics, this function requires a cell volume cell measure, unless the coordinates of the input data are regular 1D latitude and longitude coordinates so the cell volumes can be computed internally. The required supplementary variable `volcello` can be attached to the main dataset as described in *Defining supplementary variables (ancillary variables and cell measures)*.

No depth coordinate is required as this is determined by Iris.

See also *esmvalcore.preprocessor.volume_statistics()*.

10.14.3 axis_statistics

This function operates over a given axis, and removes it from the output cube.

Takes arguments:

- *axis*: direction over which the statistics will be performed. Possible values for the axis are *x*, *y*, *z*, *t*.
- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics.
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

Note: The coordinate associated to the axis over which the operation will be performed must be one-dimensional, as multidimensional coordinates are not supported in this preprocessor.

Some operations are weighted by the corresponding coordinate bounds by default. For sums, the units of the resulting cubes are multiplied by the corresponding coordinate units. See *Statistical preprocessors* for more details on supported statistics.

See also *esmvalcore.preprocessor.axis_statistics()*.

10.14.4 depth_integration

This function integrates over the depth dimension. This function does a weighted sum along the *z*-coordinate, and removes the *z* direction of the output cube. This preprocessor takes no arguments. The units of the resulting cube are multiplied by the *z*-coordinate units.

See also *esmvalcore.preprocessor.depth_integration()*.

10.14.5 extract_transect

This function extracts data along a line of constant latitude or longitude. This function takes two arguments, although only one is strictly required. The two arguments are `latitude` and `longitude`. One of these arguments needs to be set to a float, and the other can then be either ignored or set to a minimum or maximum value.

For example, if we set latitude to 0 N and leave longitude blank, it would produce a cube along the Equator. On the other hand, if we set latitude to 0 and then set longitude to `[40., 100.]` this will produce a transect of the Equator in the Indian Ocean.

See also *esmvalcore.preprocessor.extract_transect()*.

10.14.6 extract_trajectory

This function extract data along a specified trajectory. The three arguments are: `latitudes`, `longitudes` and number of point needed for extrapolation `number_points`.

If two points are provided, the `number_points` argument is used to set a the number of places to extract between the two end points.

If more than two points are provided, then `extract_trajectory` will produce a cube which has extrapolated the data of the cube to those points, and `number_points` is not needed.

Note that this function uses the expensive `interpolate` method from `Iris.analysis.trajectory`, but it may be necessary for irregular grids.

See also *esmvalcore.preprocessor.extract_trajectory()*.

10.15 Cycles

The `_cycles.py` module contains the following preprocessor functions:

- `amplitude`: Extract the peak-to-peak amplitude of a cycle aggregated over specified coordinates.

10.15.1 amplitude

This function extracts the peak-to-peak amplitude (maximum value minus minimum value) of a field aggregated over specified coordinates. Its only argument is `coords`, which can either be a single coordinate (given as `str`) or multiple coordinates (given as `list` of `str`). Usually, these coordinates refer to temporal categorised coordinates `iris.coord_categorisation` like *year*, *month*, *day of year*, etc. For example, to extract the amplitude of the annual cycle for every single year in the data, use `coords: year`; to extract the amplitude of the diurnal cycle for every single day in the data, use `coords: [year, day_of_year]`.

See also *esmvalcore.preprocessor.amplitude()*.

10.16 Trend

The trend module contains the following preprocessor functions:

- `linear_trend`: Calculate linear trend along a specified coordinate.
- `linear_trend_stderr`: Calculate standard error of linear trend along a specified coordinate.

10.16.1 linear_trend

This function calculates the linear trend of a dataset (defined as slope of an ordinary linear regression) along a specified coordinate. The only argument of this preprocessor is `coordinate` (given as `str`; default value is `'time'`).

See also *esmvalcore.preprocessor.linear_trend()*.

10.16.2 linear_trend_stderr

This function calculates the standard error of the linear trend of a dataset (defined as the standard error of the slope in an ordinary linear regression) along a specified coordinate. The only argument of this preprocessor is `coordinate` (given as `str`; default value is `'time'`). Note that the standard error is **not** identical to a confidence interval.

See also *esmvalcore.preprocessor.linear_trend_stderr()*.

10.17 Detrend

ESMValCore also supports detrending along any dimension using the preprocessor function 'detrend'. This function has two parameters:

- `dimension`: dimension to apply detrend on. Default: "time"
- `method`: It can be `linear` or `constant`. Default: `linear`

If method is `linear`, detrend will calculate the linear trend along the selected axis and subtract it to the data. For example, this can be used to remove the linear trend caused by climate change on some variables is selected dimension is time.

If method is `constant`, detrend will compute the mean along that dimension and subtract it from the data

See also *esmvalcore.preprocessor.detrend()*.

10.18 Rolling window statistics

One can calculate rolling window statistics using the preprocessor function `rolling_window_statistics`. This function takes three parameters:

- *coordinate*: Coordinate over which the rolling-window statistics is calculated.
- *operator*: Operation to apply. See *Statistical preprocessors* for more details on supported statistics.
- *window_length*: size of the rolling window to use (number of points).
- Other parameters are directly passed to the *operator* as keyword arguments. See *Statistical preprocessors* for more details.

This example applied on daily precipitation data calculates two-day rolling precipitation sum.

```
preprocessors:
  preproc_rolling_window:
    coordinate: time
    operator: sum
    window_length: 2
```

See also *esmvalcore.preprocessor.rolling_window_statistics()*.

10.19 Unit conversion

10.19.1 convert_units

Converting units is also supported. This is particularly useful in cases where different datasets might have different units, for example when comparing CMIP5 and CMIP6 variables where the units have changed or in case of observational datasets that are delivered in different units.

In these cases, having a unit conversion at the end of the processing will guarantee homogeneous input for the diagnostics.

Conversion is only supported between compatible units! In other words, converting temperature units from `degC` to `Kelvin` works fine, while changing units from `kg` to `m` will not work.

However, there are some well-defined exceptions from this rule in order to transform one quantity to another (physically related) quantity. These quantities are identified via their `standard_name` and their `units` (units convertible to the ones defined are also supported). For example, this enables conversions between precipitation fluxes measured in `kg m-2 s-1` and precipitation rates measured in `mm day-1` (and vice versa). Currently, the following special conversions are supported:

- `precipitation_flux` (`kg m-2 s-1`) – `lwe_precipitation_rate` (`mm day-1`)

Hint: Names in the list correspond to `standard_names` of the input data. Conversions are allowed from each quantity to any other quantity given in a bullet point. The corresponding target quantity is inferred from the desired target units. In addition, any other units convertible to the ones given are also supported (e.g., instead of `mm day-1`, `m s-1` is also supported).

Note: For the transformation between the different precipitation variables, a water density of `1000 kg m-3` is assumed.

See also *esmvalcore.preprocessor.convert_units()*.

10.19.2 accumulate_coordinate

This function can be used to weight data using the bounds from a given coordinate. The resulting cube will then have units given by `cube_units * coordinate_units`.

For instance, if a variable has units such as `X s-1`, using `accumulate_coordinate` on the time coordinate would result on a cube where the data would be multiplied by the time bounds and the resulting units for the variable would be converted to `X`. In this case, weighting the data with the time coordinate would allow to cancel the time units in the variable.

Note: The coordinate used to weight the data must be one-dimensional, as multidimensional coordinates are not supported in this preprocessor.

See also `esmvalcore.preprocessor.accumulate_coordinate.()`

10.20 Bias

The bias module contains the following preprocessor functions:

- `bias`: Calculate absolute or relative biases with respect to a reference dataset

10.20.1 bias

This function calculates biases with respect to a given reference dataset. For this, exactly one input dataset needs to be declared as `reference_for_bias: true` in the recipe, e.g.,

```
datasets:
  - {dataset: CanESM5, project: CMIP6, ensemble: r1i1p1f1, grid: gn}
  - {dataset: CESM2,   project: CMIP6, ensemble: r1i1p1f1, grid: gn}
  - {dataset: MIROC6,  project: CMIP6, ensemble: r1i1p1f1, grid: gn}
  - {dataset: ERA-Interim, project: OBS6, tier: 3, type: reanaly, version: 1,
     reference_for_bias: true}
```

In the example above, ERA-Interim is used as reference dataset for the bias calculation. For this preprocessor, all input datasets need to have identical dimensional coordinates. This can for example be ensured with the preprocessors *esmvalcore.preprocessor.regrid()* and/or *esmvalcore.preprocessor.regrid_time()*.

The `bias` preprocessor supports 4 optional arguments:

- `bias_type` (`str`, default: `'absolute'`): Bias type that is calculated. Can be `'absolute'` (i.e., calculate bias for dataset X and reference R as $X - R$) or `relative` (i.e, calculate bias as $\frac{X-R}{R}$).
- `denominator_mask_threshold` (`float`, default: `1e-3`): Threshold to mask values close to zero in the denominator (i.e., the reference dataset) during the calculation of relative biases. All values in the reference dataset with absolute value less than the given threshold are masked out. This setting is ignored when `bias_type` is set to `'absolute'`. Please note that for some variables with very small absolute values (e.g., carbon cycle fluxes, which are usually $< 10^{-6}$ kg m $^{-2}$ s $^{-1}$) it is absolutely essential to change the default value in order to get reasonable results.
- `keep_reference_dataset` (`bool`, default: `False`): If `True`, keep the reference dataset in the output. If `False`, drop the reference dataset.
- `exclude` (`list` of `str`): Exclude specific datasets from this preprocessor. Note that this option is only available in the recipe, not when using *esmvalcore.preprocessor.bias()* directly (e.g., in another python script). If the reference dataset has been excluded, an error is raised.

Example:

```
preprocessors:
  preproc_bias:
    bias:
      bias_type: relative
      denominator_mask_threshold: 1e-8
      keep_reference_dataset: true
      exclude: [CanESM2]
```

See also *esmvalcore.preprocessor.bias()*.

10.21 Information on maximum memory required

In the most general case, we can set upper limits on the maximum memory the analysis will require:

`Ms = (R + N) x F_eff - F_eff` - when no multi-model analysis is performed;

`Mm = (2R + N) x F_eff - 2F_eff` - when multi-model analysis is performed;

where

- `Ms`: maximum memory for non-multimodel module
- `Mm`: maximum memory for multi-model module
- `R`: computational efficiency of module; *R* is typically 2-3
- `N`: number of datasets
- `F_eff`: average size of data per dataset where `F_eff = e x f x F` where `e` is the factor that describes how lazy the data is (`e = 1` for fully realized data) and `f` describes how much the data was shrunk by the immediately previous module, e.g. time extraction, area selection or level extraction; note that for fix_data `f` relates only to the time extraction, if data is exact in time (no time selection) `f = 1` for fix_data so for cases when we deal with a lot of datasets `R + N \approx N`, data is fully realized, assuming an average size of 1.5GB for 10 years of *3D* netCDF data, `N` datasets will require:

`Ms = 1.5 x (N - 1)` GB

`Mm = 1.5 x (N - 2)` GB

As a rule of thumb, the maximum required memory at a certain time for multi-model analysis could be estimated by multiplying the number of datasets by the average file size of all the datasets; this memory intake is high but also assumes that all data is fully realized in memory; this aspect will gradually change and the amount of realized data will decrease with the increase of `dask` use.

10.22 Other

Miscellaneous functions that do not belong to any of the other categories.

10.22.1 Clip

This function clips data values to a certain minimum, maximum or range. The function takes two arguments:

- `minimum`: Lower bound of range. Default: `None`
- `maximum`: Upper bound of range. Default: `None`

The example below shows how to set all values below zero to zero.

```
preprocessors:
  clip:
    minimum: 0
    maximum: null
```

Part IV

Diagnostic script interfaces

In order to communicate with diagnostic scripts, ESMValCore uses YAML files. The YAML files provided by ESMValCore to the diagnostic script tell the diagnostic script the settings that were provided in the recipe and where to find the pre-processed input data. On the other hand, the YAML file provided by the diagnostic script to ESMValCore tells ESMValCore which pre-processed data was used to create what plots. The latter is optional, but needed for recording provenance.

CHAPTER

ELEVEN

PROVENANCE

When ESMValCore (the `esmvaltool` command) runs a recipe, it will first find all data and run the default preprocessor steps plus any additional preprocessing steps defined in the recipe. Next it will run the diagnostic script defined in the recipe and finally it will store provenance information. Provenance information is stored in the W3C PROV XML format. To read in and extract information, or to plot these files, the prov Python package can be used. In addition to provenance information, a caption is also added to the plots.

CHAPTER

TWELVE

INFORMATION PROVIDED BY ESMVALCORE TO THE DIAGNOSTIC SCRIPT

To provide the diagnostic script with the information it needs to run (e.g. location of input data, various settings), the ESMValCore creates a YAML file called settings.yml and provides the path to this file as the first command line argument to the diagnostic script.

The most interesting settings provided in this file are

```
run_dir:  /path/to/recipe_output/run/diagnostic_name/script_name
work_dir: /path/to/recipe_output/work/diagnostic_name/script_name
plot_dir: /path/to/recipe_output/plots/diagnostic_name/script_name
input_files:
  - /path/to/recipe_output/preproc/diagnostic_name/ta/metadata.yml
  - /path/to/recipe_output/preproc/diagnostic_name/pr/metadata.yml
```

Custom settings in the script section of the recipe will also be made available in this file.

There are three directories defined:

- `run_dir` use this for storing temporary files
- `work_dir` use this for storing NetCDF files containing the data used to make a plot
- `plot_dir` use this for storing plots

Finally `input_files` is a list of YAML files, containing a description of the preprocessed data. Each entry in these YAML files is a path to a preprocessed file in NetCDF format, with a list of various attributes. An example preprocessor metadata.yml file could look like this:

```
? /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_historical_
↪r1i1p1_T2Ms_pr_2000-2002.nc
: alias: GFDL-ESM2G
  cmor_table: CMIP5
  dataset: GFDL-ESM2G
  diagnostic: diagnostic_name
  end_year: 2002
  ensemble: r1i1p1
  exp: historical
  filename: /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc
  frequency: mon
  institute: [NOAA-GFDL]
  long_name: Precipitation
  mip: Amon
```

(continues on next page)

(continued from previous page)

```
  modeling_realm: [atmos]
  preprocessor: preprocessor_name
  project: CMIP5
  recipe_dataset_index: 1
  reference_dataset: MPI-ESM-LR
  short_name: pr
  standard_name: precipitation_flux
  start_year: 2000
  units: kg m-2 s-1
  variable_group: pr
? /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_MPI-ESM-LR_Amon_historical_
↪r1i1p1_T2Ms_pr_2000-2002.nc
: alias: MPI-ESM-LR
  cmor_table: CMIP5
  dataset: MPI-ESM-LR
  diagnostic: diagnostic_name
  end_year: 2002
  ensemble: r1i1p1
  exp: historical
  filename: /path/to/recipe_output/preproc/diagnostic1/pr/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc
  frequency: mon
  institute: [MPI-M]
  long_name: Precipitation
  mip: Amon
  modeling_realm: [atmos]
  preprocessor: preprocessor_name
  project: CMIP5
  recipe_dataset_index: 2
  reference_dataset: MPI-ESM-LR
  short_name: pr
  standard_name: precipitation_flux
  start_year: 2000
  units: kg m-2 s-1
  variable_group: pr
```

CHAPTER

THIRTEEN

INFORMATION PROVIDED BY THE DIAGNOSTIC SCRIPT TO ESMVALCORE

After the diagnostic script has finished running, ESMValCore will try to store provenance information. In order to link the produced files to input data, the diagnostic script needs to store a YAML file called `diagnostic_provenance.yml` in its `run_dir`.

For every output file (netCDF files, plot files, etc.) produced by the diagnostic script, there should be an entry in the `diagnostic_provenance.yml` file. The name of each entry should be the path to the file. Each output file entry should at least contain the following items:

- `ancestors` a list of input files used to create the plot.
- `caption` a caption text for the plot.

Each file entry can also contain items from the categories defined in the file `esmvaltool/config_references.yml`. The short entries will automatically be replaced by their longer equivalent in the final provenance records. It is possible to add custom provenance information by adding custom items to entries.

An example `diagnostic_provenance.yml` file could look like this

```
? /path/to/recipe_output/work/diagnostic_name/script_name/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_pr_2000-2002_mean.nc
: ancestors:[/path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_pr_2000-2002.nc]
  authors: [andela_bouwe, righi_mattia]
  caption: Average Precipitation between 2000 and 2002 according to GFDL-ESM2G.
  domains: [global]
  plot_types: [zonal]
  references: [acknow_project]
  statistics: [mean]

? /path/to/recipe_output/plots/diagnostic_name/script_name/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_pr_2000-2002_mean.png
: ancestors:[/path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_pr_2000-2002.nc]
  authors: [andela_bouwe, righi_mattia]
  caption: Average Precipitation between 2000 and 2002 according to GFDL-ESM2G.
  domains: [global]
  plot_types: ['zonal']
  references: [acknow_project]
  statistics: [mean]
```

You can check whether your diagnostic script successfully provided the provenance information to the ESMValCore by checking the following points:

- for each output file in the `work_dir` and `plot_dir`, a file with the same name, but ending with `_provenance.xml` is created
- the output file is shown on the `index.html` page
- there were no warning messages in the log related to provenance

See Recording provenance for more extensive usage notes.

Part V

Development

To get started developing, have a look at our *contribution guidelines*. This chapter describes how to implement the most commonly contributed new features.

CHAPTER

FOURTEEN

PREPROCESSOR FUNCTION

Preprocessor functions are located in *esmvalcore.preprocessor*. To add a new preprocessor function, start by finding a likely looking file to add your function to in esmvalcore/preprocessor. Create a new file in that directory if you cannot find a suitable place.

The function should look like this:

```
def example_preprocessor_function(
    cube,
    example_argument,
    example_optional_argument=5,
):
    """Compute an example quantity.

    A more extensive explanation of the computation can be added here. Add
    references to scientific literature if available.

    Parameters
    ----------
    cube: iris.cube.Cube
       Input cube.

    example_argument: str
       Example argument, the value of this argument can be provided in the
       recipe. Describe what valid values are here. In this case, a valid
       argument is the name of a dimension of the input cube.

    example_optional_argument: int, optional
       Another example argument, the value of this argument can optionally
       be provided in the recipe. Describe what valid values are here.

    Returns
    -------
    iris.cube.Cube
      The result of the example computation.
    """

    # Replace this with your own computation
    cube = cube.collapsed(example_argument, iris.analysis.MEAN)

    return cube
```

The above function needs to be imported in the file esmvalcore/preprocessor/__init__.py:

```
from ._example_module import example_preprocessor_function

__all__ = [
...
'example_preprocessor_function',
...
]
```

The location in the `__all__` list above determines the default order in which preprocessor functions are applied, so carefully consider where you put it and ask for advice if needed.

The preprocessor function above can then be used from the *Recipe section: preprocessors* like this:

```
preprocessors:
  example_preprocessor:
    example_preprocessor_function:
      example_argument: median
      example_optional_argument: 6
```

The optional argument (in this example: `example_optional_argument`) can be omitted in the recipe.

14.1 Lazy and real data

Preprocessor functions should support both real and lazy data. This is vital for supporting the large datasets that are typically used with the ESMValCore. If the data of the incoming cube has been realized (i.e. `cube.has_lazy_data()` returns `False` so `cube.core_data()` is a NumPy array), the returned cube should also have realized data. Conversely, if the incoming cube has lazy data (i.e. `cube.has_lazy_data()` returns `True` so `cube.core_data()` is a Dask array), the returned cube should also have lazy data. Note that NumPy functions will often call their Dask equivalent if it exists and if their input array is a Dask array, and vice versa.

Note that preprocessor functions should preferably be small and just call the relevant iris code. Code that is more involved, e.g. lots of work with Numpy and Dask arrays, and more broadly applicable, should be implemented in iris instead.

14.2 Metadata

Preprocessor functions may change the metadata of datasets. An obvious example is *`convert_units()`*, which changes units. If cube metadata is changed in a preprocessor function, the *metadata.yml* file is automatically updated with this information. The following attributes are taken into account:

Attribute in `metadata.yml` file	Updated from
`standard_name`	`iris.cube.Cube.standard_name`
`long_name`	`iris.cube.Cube.long_name`
`short_name`	`iris.cube.Cube.var_name`
`units`	`iris.cube.Cube.units`
`frequency`	`iris.cube.Cube.attributes['frequency']`

If a given cube property is `None`, the corresponding attribute is updated with an empty string (`''`). If a cube property is not given, the corresponding attribute is not updated.

14.3 Documentation

The documentation in the function docstring will be shown in the *Preprocessor functions* chapter. In addition, you should add documentation on how to use the new preprocessor function from the recipe in doc/recipe/preprocessor.rst so it is shown in the *Preprocessor* chapter. See the introduction to *Documentation* for more information on how to best write documentation.

14.4 Tests

Tests are should be implemented for new or modified preprocessor functions. For an introduction to the topic, see *Tests*.

14.4.1 Unit tests

To add a unit test for the preprocessor function from the example above, create a file called `tests/unit/preprocessor/_example_module/test_example_preprocessor_function.py` and add the following content:

```
"""Test function `esmvalcore.preprocessor.example_preprocessor_function`."""
import cf_units
import dask.array as da
import iris
import numpy as np
import pytest

from esmvalcore.preprocessor import example_preprocessor_function


@pytest.mark.parametrize('lazy', [True, False])
def test_example_preprocessor_function(lazy):
    """Test that the computed result is as expected."""

    # Construct the input cube
    data = np.array([1, 2], dtype=np.float32)
    if lazy:
        data = da.asarray(data, chunks=(1, ))
    cube = iris.cube.Cube(
        data,
        var_name='tas',
        units='K',
    )
    cube.add_dim_coord(
        iris.coords.DimCoord(
            np.array([0.5, 1.5], dtype=np.float64),
            bounds=np.array([[0, 1], [1, 2]], dtype=np.float64),
            standard_name='time',
            units=cf_units.Unit('days since 1950-01-01 00:00:00',
                                calendar='gregorian'),
        ),
        0,
    )
```

(continues on next page)

(continued from previous page)

```
    # Compute the result
    result = example_preprocessor_function(cube, example_argument='time')

    # Check that lazy data is returned if and only if the input is lazy
    assert result.has_lazy_data() is lazy

    # Construct the expected result cube
    expected = iris.cube.Cube(
        np.array(1.5, dtype=np.float32),
        var_name='tas',
        units='K',
    )
    expected.add_aux_coord(
        iris.coords.AuxCoord(
            np.array([1], dtype=np.float64),
            bounds=np.array([[0, 2]], dtype=np.float64),
            standard_name='time',
            units=cf_units.Unit('days since 1950-01-01 00:00:00',
                                calendar='gregorian'),
        ))
    expected.add_cell_method(
        iris.coords.CellMethod(method='mean', coords=('time', )))

    # Compare the result of the computation with the expected result
    print('result:', result)
    print('expected result:', expected)
    assert result == expected
```

In this test we used the decorator pytest.mark.parametrize to test two scenarios, with both lazy and realized data, with a single test.

14.4.2 Sample data tests

The idea of adding *sample data tests* is to check that preprocessor functions work with realistic data. This also provides an easy way to add regression tests, though these should preferably be implemented as unit tests instead, because using the sample data for this purpose is slow. To add a test using the sample data, create a file `tests/sample_data/preprocessor/example_preprocessor_function/test_example_preprocessor_function.py` and add the following content:

```
"""Test function `esmvalcore.preprocessor.example_preprocessor_function`."""
from pathlib import Path

import esmvaltool_sample_data
import iris
import pytest

from esmvalcore.preprocessor import example_preprocessor_function


@pytest.mark.use_sample_data
def test_example_preprocessor_function():
```

(continues on next page)

(continued from previous page)

```
    """Regression test to check that the computed result is as expected."""
    # Load an example input cube
    cube = esmvaltool_sample_data.load_timeseries_cubes(mip_table='Amon')[0]

    # Compute the result
    result = example_preprocessor_function(cube, example_argument='time')

    filename = Path(__file__).with_name('example_preprocessor_function.nc')
    if not filename.exists():
        # Create the file the expected result if it doesn't exist
        iris.save(result, target=str(filename))
        raise FileNotFoundError(
            f'Reference data was missing, wrote new copy to {filename}')

    # Load the expected result cube
    expected = iris.load_cube(str(filename))

    # Compare the result of the computation with the expected result
    print('result:', result)
    print('expected result:', expected)
    assert result == expected
```

This will use a file from the sample data repository as input. The first time you run the test, the computed result will be stored in the file `tests/sample_data/preprocessor/example_preprocessor_function/example_preprocessor_function.nc` Any subsequent runs will re-load the data from file and check that it did not change. Make sure the stored results are small, i.e. smaller than 100 kilobytes, to keep the size of the ESMValCore repository small.

14.5 Using multiple datasets as input

The name of the first argument of the preprocessor function should in almost all cases be `cube`. Only when implementing a preprocessor function that uses all datasets as input, the name of the first argument should be `products`. If you would like to implement this type of preprocessor function, start by having a look at the existing functions, e.g. *esmvalcore.preprocessor.multi_model_statistics()* or *esmvalcore.preprocessor.mask_fillvalues()*.

CHAPTER

FIFTEEN

FIXING DATA

The baseline case for ESMValCore input data is CMOR fully compliant data that is read using Iris' `iris.load_raw()`. ESMValCore also allows for some departures from compliance (see *Customizing checker strictness*). Beyond that situation, some datasets (either model or observations) contain (known) errors that would normally prevent them from being processed. The issues can be in the metadata describing the dataset and/or in the actual data. Typical examples of such errors are missing or wrong attributes (e.g. attribute ''units" says 1e-9 but data are actually in 1e-6), missing or mislabeled coordinates (e.g. ''lev" instead of ''plev" or missing coordinate bounds like ''lat_bnds") or problems with the actual data (e.g. cloud liquid water only instead of sum of liquid + ice as specified by the CMIP data request). As an extreme case, some data sources simply are not NetCDF files and must go through some other data load function. ESMValCore can apply on-the-fly fixes to such datasets when issues can be fixed automatically.

In addition, some datasets are supported in their native (i.e., non CMOR-compliant) format through fixes. This is implemented for a set of *Datasets in native format*. A detailed description of how to include new native datasets is given *below*.

The following sections provide details on how to design such fixes.

Note: CMORizer scripts. Support for many observational and reanalysis datasets is also possible through a priori reformatting by CMORizer scripts in the ESMValTool, which are rather relevant for datasets of small volume

---

15.1 Fix structure

Fixes are Python classes stored in `esmvalcore/cmor/_fixes/[PROJECT]/[DATASET].py` that derive from `esmvalcore.cmor._fixes.fix.Fix` and are named after the short name of the variable they fix. You can also use the names of `mip` tables (e.g., `Amon`, `Lmon`, `Omon`, etc.) if you want the fix to be applied to all variables of that table in the dataset or `AllVars` if you want the fix to be applied to the whole dataset.

Warning: Be careful to replace any `-` with `_` in your dataset name. We need this replacement to have proper python module names.

The fixes are automatically loaded and applied when the dataset is preprocessed. They are a special type of *preprocessor function*, called by the preprocessor functions *esmvalcore.preprocessor.fix_file()*, *esmvalcore.preprocessor.fix_metadata()*, and *esmvalcore.preprocessor.fix_data()*.

The `Fix` class provides the following attributes:

- `Fix.vardef`: *VariableInfo* object that corresponds to the variable fixed by the fix.
- `Fix.extra_facets`: `dict` that contains all facets of the corresponding dataset fixed by the fix (see *esmvalcore.dataset.Dataset.facets*).

- `Fix.session`: *Session* object that includes configuration and directory information.

15.2 Fixing a dataset

To illustrate the process of creating a fix we are going to construct a new one from scratch for a fictional dataset. We need to fix a CMIPX model called PERFECT-MODEL that is reporting a missing latitude coordinate for variable tas.

15.2.1 Check the output

Next to the error message, you should see some info about the iris cube: size, coordinates. In our example it looks like this:

```
air_temperature/ (K) (time: 312; altitude: 90; longitude: 180)
    Dimension coordinates:
        time                                    x              -              -
        altitude                                -              x              -
        longitude                               -              -              x
    Auxiliary coordinates:
        day_of_month                            x              -              -
        day_of_year                             x              -              -
        month_number                            x              -              -
        year                                    x              -              -
    Attributes:
        {'cmor_table': 'CMIPX', 'mip': 'Amon', 'short_name': 'tas', 'frequency': 'mon'})
```

So now the mistake is clear: the latitude coordinate is badly named and the fix should just rename it.

15.2.2 Create the fix

We start by creating the module file. In our example the path will be `esmvalcore/cmor/_fixes/CMIPX/PERFECT_MODEL.py`. If it already exists just add the class to the file, there is no limit in the number of fixes we can have in any given file.

Then we have to create the class for the fix deriving from `esmvalcore.cmor._fixes.Fix`

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
     """Fixes for tas variable."""
```

Next we must choose the method to use between the ones offered by the Fix class:

- `fix_file`: should be used only to fix errors that prevent data loading. As a rule of thumb, you should only use it if the execution halts before reaching the checks.
- `fix_metadata`: you want to change something in the cube that is not the data (e.g., variable or coordinate names, data units).
- `fix_data`: you need to fix the data. Beware: coordinates data values are part of the metadata.

In our case we need to rename the coordinate `altitude` to `latitude`, so we will implement the `fix_metadata` method:

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
    """Fixes for tas variable."""

    def fix_metadata(self, cubes):
        """
        Fix metadata for tas.

        Fix the name of the latitude coordinate, which is called altitude
        in the original file.
        """
        # Sometimes Iris will interpret the data as multiple cubes.
        # Good CMOR datasets will only show one but we support the
        # multiple cubes case to be able to fix the errors that are
        # leading to that extra cubes.
        # In our case this means that we can safely assume that the
        # tas cube is the first one
        tas_cube = cubes[0]
        latitude = tas_cube.coord('altitude')

        # Fix the names. Latitude values, units and
        latitude.short_name = 'lat'
        latitude.standard_name = 'latitude'
        latitude.long_name = 'latitude'
        return cubes
```

This will fix the error. The next time you run ESMValTool you will find that the error is fixed on the fly and, hopefully, your recipe will run free of errors. The `cubes` argument to the `fix_metadata` method will contain all cubes loaded from a single input file. Some care may need to be taken that the right cube is selected and fixed in case multiple cubes are created. Usually this happens when a coordinate is mistakenly loaded as a cube, because the input data does not follow the CF Conventions.

Sometimes other errors can appear after you fix the first one because they were hidden by it. In our case, the latitude coordinate could have bad units or values outside the valid range for example. Just extend your fix to address those errors.

15.2.3 Finishing

Chances are that you are not the only one that wants to use that dataset and variable. Other users could take advantage of your fixes as soon as possible. Please, create a separated pull request for the fix and submit it.

It will also be very helpful if you just scan a couple of other variables from the same dataset and check if they share this error. In case that you find that it is a general one, you can change the fix name to the corresponding `mip` table name (e.g., `Amon`, `Lmon`, `Omon`, etc.) so it gets executed for all variables in that table in the dataset or to `AllVars` so it gets executed for all variables in the dataset. If you find that this is shared only by a handful of similar vars you can just make the fix for those new vars derive from the one you just created:

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
```

(continues on next page)

(continued from previous page)

```
    """Fixes for tas variable."""

    def fix_metadata(self, cubes):
        """
        Fix metadata for tas.

        Fix the name of the latitude coordinate, which is called altitude
        in the original file.
        """
        # Sometimes Iris will interpret the data as multiple cubes.
        # Good CMOR datasets will only show one but we support the
        # multiple cubes case to be able to fix the errors that are
        # leading to that extra cubes.
        # In our case this means that we can safely assume that the
        # tas cube is the first one
        tas_cube = cubes[0]
        latitude = tas_cube.coord('altitude')

        # Fix the names. Latitude values, units and
        latitude.short_name = 'lat'
        latitude.standard_name = 'latitude'
        latitude.long_name = 'latitude'
        return cubes


class ps(tas):
    """Fixes for ps variable."""
```

15.3 Common errors

The above example covers one of the most common cases: variables / coordinates that have names that do not match the expected. But there are some others that use to appear frequently. This section describes the most common cases.

15.3.1 Bad units declared

It is quite common that a variable declares to be using some units but the data is stored in another. This can be solved by overwriting the units attribute with the actual data units.

```
def fix_metadata(self, cubes):
    cube = self.get_cube_from_list(cubes)
    cube.units = 'real_units'
```

Detecting this error can be tricky if the units are similar enough. It also has a good chance of going undetected until you notice strange results in your diagnostic.

For the above example, it can be useful to access the variable definition and associated coordinate definitions as provided by the CMOR table. For example:

```
def fix_metadata(self, cubes):
    cube = self.get_cube_from_list(cubes)
    cube.units = self.vardef.units
```

To learn more about what is available in these definitions, see: *esmvalcore.cmor.table.VariableInfo* and *esmvalcore.cmor.table.CoordinateInfo*.

15.3.2 Coordinates missing

Another common error is to have missing coordinates. Usually it just means that the file does not follow the CF-conventions and Iris can therefore not interpret it.

If this is the case, you should see a warning from the ESMValTool about discarding some cubes in the fix metadata step. Just before that warning you should see the full list of cubes as read by Iris. If that list contains your missing coordinate you can create a fix for this model:

```
def fix_metadata(self, cubes):
    coord_cube = cubes.extract_cube('COORDINATE_NAME')
    # Usually this will correspond to an auxiliary coordinate
    # because the most common error is to forget adding it to the
    # coordinates attribute
    coord = iris.coords.AuxCoord(
        coord_cube.data,
        var_name=coord_cube.var_name,
        standard_name=coord_cube.standard_name,
        long_name=coord_cube.long_name,
        units=coord_cube.units,
    }

    # It may also have bounds as another cube
    coord.bounds = cubes.extract_cube('BOUNDS_NAME').data

    data_cube = cubes.extract_cube('VAR_NAME')
    data_cube.add_aux_coord(coord, DIMENSIONS_INDEX_TUPLE)
    return [data_cube]
```

15.4 Customizing checker strictness

The data checker classifies its issues using four different levels of severity. From highest to lowest:

- `CRITICAL`: issues that most of the time will have severe consequences.
- `ERROR`: issues that usually lead to unexpected errors, but can be safely ignored sometimes.
- `WARNING`: something is not up to the standard but is unlikely to have consequences later.
- `DEBUG`: any info that the checker wants to communicate. Regardless of checker strictness, those will always be reported as debug messages.

Users can have control about which levels of issues are interpreted as errors, and therefore make the checker fail or warnings or debug messages. For this purpose there is an optional command line option *–check-level* that can take a number of values, listed below from the lowest level of strictness to the highest:

- `ignore`: all issues, regardless of severity, will be reported as warnings. Checker will never fail. Use this at your own risk.
- `relaxed`: only CRITICAL issues are treated as errors. We recommend not to rely on this mode, although it can be useful if there are errors preventing the run that you are sure you can manage on the diagnostics or that will not affect you.
- `default`: fail if there are any CRITICAL or ERROR issues (DEFAULT); Provides a good measure of safety.
- `strict`: fail if there are any warnings, this is the highest level of strictness. Mostly useful for checking datasets that you have produced, to be sure that future users will not be distracted by inoffensive warnings.

15.5 Add support for new native datasets

This section describes how to add support for additional native datasets. You can choose to host this new data source either under a dedicated project or under project `native6`.

15.5.1 Configuration

An example of a configuration in `config-developer.yml` for projects used for native datasets is given *here*. Make sure to use the option `cmor_strict: false` for these projects if you want to make use of *Custom CMOR tables*. This allows reading arbitrary variables from native datasets.

15.5.2 Locate data

To allow ESMValCore to locate the data files, use the following steps:

- If you want to use the `native6` project (recommended for datasets whose input files can be easily moved to the usual `native6` directory structure given by the `rootpath` in your *User configuration file*; this is usually the case for native reanalysis/observational datasets):

  The entry `native6` of `config-developer.yml` should be complemented with sub-entries for `input_dir` and `input_file` that go under a new key representing the data organization (such as `MY_DATA_ORG`), and these sub-entries can use an arbitrary list of `{placeholders}`. Example :

```
native6:
  ...
  input_dir:
    default: 'Tier{tier}/{dataset}/{version}/{frequency}/{short_name}'
    MY_DATA_ORG: '{dataset}/{exp}/{simulation}/{version}/{type}'
  input_file:
    default: '*.nc'
    MY_DATA_ORG: '{simulation}_*.nc'
  ...
```

  To find your native data (e.g., called `MYDATA`) that is for example located in `{rootpath}/MYDATA/amip/run1/42-0/atm/run1_1979.nc` (`{rootpath}` is ESMValTool's `rootpath` for the project `native6` defined in your *User configuration file*), use the following dataset entry in your recipe

```
datasets:
  - {project: native6, dataset: MYDATA, exp: amip, simulation: run1, version: 42-0,␣
↪type: atm}
```

and make sure to use the following DRS for the project `native6` in your *User configuration file*:

```
drs:
  native6: MY_DATA_ORG
```

- If you want to use a dedicated project for your native dataset (recommended for datasets for which you cannot control the location of the input files; this is usually the case for native model output):

  A new entry for the project needs to be added to `config-developer.yml`. For example, for the ICON model, create a new project `ICON`:

```
ICON:
  ...
  input_dir:
    default:
      - '{exp}'
      - '{exp}/outdata'
  input_file:
    default: '{exp}_{var_type}*.nc'
  ...
```

  To find your ICON data that is for example located in files like `{rootpath}/amip/amip_atm_2d_ml_20000101T000000Z.nc` (`{rootpath}` is ESMValTool `rootpath` for the project `ICON` defined in your *User configuration file*), use the following dataset entry in your recipe:

```
datasets:
  - {project: ICON, dataset: ICON, exp: amip}
```

  Please note the duplication of the name `ICON` in `project` and `dataset`, which is necessary to comply with ESMValTool's data finding and CMORizing functionalities. For other native models, `dataset` could also refer to a subversion of the model. Note that it is possible to predefine facets in an *extra facets file*. In this ICON example, the facet `var_type` is `predefined` for many variables.

15.5.3 Fix native data

To ensure that the native dataset has the correct metadata and data (i.e., that it is CMOR-compliant), use *dataset fixes*. This is where the actual CMORization takes place. For example, a `native6` dataset fix for ERA5 is located here, and the `ICON` fix is located here.

ESMValTool also provides a base class `NativeDatasetFix` that provides convenient functions useful for all native dataset fixes. An example for its usage can be found here.

15.5.4 Extra facets for native datasets

If necessary, provide a so-called `extra facets file` which allows to cope e.g. with variable naming issues for finding files or additional information that is required for the fixes. See *Extra Facets* and *Use of extra facets in fixes* for more details on this. An example of such a file for IPSL-CM6 is given `here`.

15.6 Use of extra facets in fixes

Extra facets are a mechanism to provide additional information for certain kinds of data. The general approach is described in *Extra Facets*. Here, we describe how they can be used in fixes to mold data into the form required by the applicable standard. For example, if the input data is part of an observational product that delivers surface temperature with a variable name of *t2m* inside a file named *2m_temperature_1950_monthly.nc*, but the same variable is called *tas* in the applicable standard, a fix can be created that reads the original variable from the correct file, and provides a renamed variable to the rest of the processing chain.

Normally, the applicable standard for variables is CMIP6.

For more details, refer to existing uses of this feature as examples, as e.g. *for IPSL-CM6*.

CHAPTER

SIXTEEN

DERIVING A VARIABLE

The variable derivation preprocessor module allows to derive variables which are not in the CMIP standard data request using standard variables as input. This is a special type of *preprocessor function*. All derivation scripts are located in esmvalcore/preprocessor/_derive/. A typical example looks like this:

```
"""Derivation of variable `dummy`."""
from ._baseclass import DerivedVariableBase


class DerivedVariable(DerivedVariableBase):
    """Derivation of variable `dummy`."""

    @staticmethod
    def required(project):
        """Declare the variables needed for derivation."""
        mip = 'fx'
        if project == 'CMIP6':
            mip = 'Ofx'
        required = [
            {'short_name': 'var_a'},
            {'short_name': 'var_b', 'mip': mip, 'optional': True},
        ]
        return required

    @staticmethod
    def calculate(cubes):
        """Compute `dummy`."""

        # `cubes` is a CubeList containing all required variables.
        cube = do_something_with(cubes)

        # Return single cube at the end
        return cube
```

The static function `required(project)` returns a `list` of `dict` containing all required variables for deriving the derived variable. Its only argument is the `project` of the specific dataset. In this particular example script, the derived variable `dummy` is derived from `var_a` and `var_b`. It is possible to specify arbitrary attributes for each required variable, e.g. `var_b` uses the mip `fx` (or `Ofx` in the case of CMIP6) instead of the original one of `dummy`. Note that you can also declare a required variable as `optional=True`, which allows the skipping of this particular variable during data extraction. For example, this is useful for fx variables which are often not available for observational datasets. Otherwise, the tool will fail if not all required variables are available for all datasets.

The actual derivation takes place in the static function `calculate(cubes)` which returns a single `cube` containing

the derived variable. Its only argument `cubes` is a `CubeList` containing all required variables.

Part VI

Contributions are very welcome

We greatly value contributions of any kind. Contributions could include, but are not limited to documentation improvements, bug reports, new or improved code, scientific and technical code reviews, infrastructure improvements, mailing list and chat participation, community help/building, education and outreach. We value the time you invest in contributing and strive to make the process as easy as possible. If you have suggestions for improving the process of contributing, please do not hesitate to propose them.

If you have a bug or other issue to report or just need help, please open an issue on the issues tab on the ESMValCore github repository.

If you would like to contribute a new *preprocessor function*, *derived variable*, *fix for a dataset*, or another new feature, please discuss your idea with the development team before getting started, to avoid double work and/or disappointment later. A good way to do this is to open an issue on GitHub.

CHAPTER

SEVENTEEN

GETTING STARTED

See *Installation from source* for instructions on how to set up a development installation.

New development should preferably be done in the ESMValCore GitHub repository. The default git branch is `main`. Use this branch to create a new feature branch from and make a pull request against. This page offers a good introduction to git branches, but it was written for BitBucket while we use GitHub, so replace the word BitBucket by GitHub whenever you read it.

It is recommended that you open a draft pull request early, as this will cause *CircleCI to run the unit tests*, *Codacy to analyse your code*, and *readthedocs to build the documentation*. It's also easier to get help from other developers if your code is visible in a pull request.

Make small pull requests, the ideal pull requests changes just a few files and adds/changes no more than 100 lines of production code. The amount of test code added can be more extensive, but changes to existing test code should be made sparingly.

17.1 Design considerations

When making changes, try to respect the current structure of the program. If you need to make major changes to the structure of program to add a feature, chances are that you have either not come up with the most optimal design or the feature is not a very good fit for the tool. Discuss your feature with the @ESMValGroup/esmvaltool-coreteam in an issue to find a solution.

Please keep the following considerations in mind when programming:

- Changes should preferably be *backward compatible*.
- Apply changes gradually and change no more than a few files in a single pull request, but do make sure every pull request in itself brings a meaningful improvement. This reduces the risk of breaking existing functionality and making *backward incompatible* changes, because it helps you as well as the reviewers of your pull request to better understand what exactly is being changed.
- *Preprocessor functions* are Python functions (and not classes) so they are easy to understand and implement for scientific contributors.
- No additional CMOR checks should be implemented inside preprocessor functions. The input cube is fixed and confirmed to follow the specification in esmvalcore/cmor/tables before applying any other preprocessor functions. This design helps to keep the preprocessor functions and diagnostics scripts that use the preprocessed data from the tool simple and reliable. See *Project CMOR table configuration* for the mapping from `project` in the recipe to the relevant CMOR table.
- The ESMValCore package is based on iris. Preprocessor functions should preferably be small and just call the relevant iris code. Code that is more involved and more broadly applicable than just in the ESMValCore, should be implemented in iris instead.

- Any settings in the recipe that can be checked before loading the data should be checked at the *task creation stage*. This avoids that users run a recipe for several hours before finding out they made a mistake in the recipe. No data should be processed or files written while creating the tasks.
- CMOR checks should provide a good balance between reliability of the tool and ease of use. Several *levels of strictness of the checks* are available to facilitate this.
- Keep your code short and simple: we would like to make contributing as easy as possible. For example, avoid implementing complicated class inheritance structures and boilerplate code.
- If you find yourself copy-pasting a piece of code and making minor changes to every copy, instead put the repeated bit of code in a function that you can re-use, and provide the changed bits as function arguments.
- Be careful when changing existing unit tests to make your new feature work. You might be breaking existing features if you have to change existing tests.

Finally, if you would like to improve the design of the tool, discuss your plans with the @ESMValGroup/esmvaltool-coreteam to make sure you understand the current functionality and you all agree on the new design.

CHAPTER

EIGHTEEN

CHECKLIST FOR PULL REQUESTS

To clearly communicate up front what is expected from a pull request, we have the following checklist. Please try to do everything on the list before requesting a review. If you are unsure about something on the list, please ask the @ESMValGroup/tech-reviewers or @ESMValGroup/science-reviewers for help by commenting on your (draft) pull request or by starting a new discussion.

In the ESMValTool community we use pull request reviews to ensure all code and documentation contributions are of good quality. The icons indicate whether the item will be checked during the Technical review or Scientific review.

- The new functionality is *relevant and scientifically sound*
- *The pull request has a descriptive title and labels*
- Code is written according to the *code quality guidelines*
- and *Documentation* is available
- Unit *tests* have been added
- Changes are *backward compatible*
- Changed *dependencies have been added or removed correctly*
- The *list of authors* is up to date
- The *checks shown below the pull request* are successful

Pull requests introducing a change that causes a recipe to no longer run successfully (*breaking change*), or which results in scientifically significant changes in results (*science change*) require additional items to be reviewed defined in the backward compatibility policy. These include in particular:

- Instructions for the release notes to assist *recipe developers* to adapt their recipe in light of the *backward-incompatible change* available.
- General instructions for *recipe developers* working on *user recipes* to enable them to adapt their code related to *backward-incompatible changes* available (see ESMValTool_Tutorial: issue #263).
- Core development team tagged to notify them of the *backward-incompatible change*, and give at least 2 weeks for objections to be raised before merging to the main branch. If a strong objection is raised the backward-incompatible change should not be merged until the objection is resolved.
- Information required for the "*backward-incompatible changes*" section in the PR that introduces the *backward-incompatible change* available.

CHAPTER

NINETEEN

SCIENTIFIC RELEVANCE

The proposed changes should be relevant for the larger scientific community. The implementation of new features should be scientifically sound; e.g. the formulas used in new preprocesssor functions should be accompanied by the relevant references and checked for correctness by the scientific reviewer. The CF Conventions as well as additional standards imposed by CMIP should be followed whenever possible.

CHAPTER

TWENTY

PULL REQUEST TITLE AND LABEL

The title of a pull request should clearly describe what the pull request changes. If you need more text to describe what the pull request does, please add it in the description. Add one or more labels to your pull request to indicate the type of change. At least one of the following labels should be used: *bug*, *deprecated feature*, *fix for dataset*, *preprocessor*, *cmor*, *api*, *testing*, *documentation* or *enhancement*.

The titles and labels of pull requests are used to compile the *Changelog*, therefore it is important that they are easy to understand for people who are not familiar with the code or people in the project. Descriptive pull request titles also makes it easier to find back what was changed when, which is useful in case a bug was introduced.

CHAPTER

TWENTYONE

CODE QUALITY

To increase the readability and maintainability or the ESMValCore source code, we aim to adhere to best practices and coding standards.

We include checks for Python and yaml files, most of which are described in more detail in the sections below. This includes checks for invalid syntax and formatting errors. Pre-commit is a handy tool that can run all of these checks automatically just before you commit your code. It knows knows which tool to run for each filetype, and therefore provides a convenient way to check your code.

21.1 Python

The standard document on best practices for Python code is PEP8 and there is PEP257 for code documentation. We make use of numpy style docstrings to document Python functions that are visible on readthedocs.

To check if your code adheres to the standard, go to the directory where the repository is cloned, e.g. `cd ESMValCore`, and run prospector

```
prospector esmvalcore/preprocessor/_regrid.py
```

In addition to prospector, we use flake8 to automatically check for bugs and formatting mistakes and mypy for checking that type hints are correct. Note that type hints are completely optional, but if you do choose to add them, they should be correct.

When you make a pull request, adherence to the Python development best practices is checked in two ways:

1. As part of the unit tests, flake8 and mypy are run by CircleCI, see the section on *Tests* for more information.
2. Codacy is a service that runs prospector (and other code quality tools) on changed files and reports the results. Click the 'Details' link behind the Codacy check entry and then click 'View more details on Codacy Production' to see the results of the static code analysis done by Codacy. If you need to log in, you can do so using your GitHub account.

The automatic code quality checks by prospector are really helpful to improve the quality of your code, but they are not flawless. If you suspect prospector or Codacy may be wrong, please ask the @ESMValGroup/tech-reviewers by commenting on your pull request.

Note that running prospector locally will give you quicker and sometimes more accurate results than waiting for Codacy.

Most formatting issues in Python code can be fixed automatically by running the commands

```
isort some_file.py
```

to sort the imports in the standard way using isort and

```
yapf -i some_file.py
```

to add/remove whitespace as required by the standard using yapf,

```
docformatter -i some_file.py
```

to run docformatter which helps formatting the docstrings (such as line length, spaces).

21.2 YAML

Please use yamllint to check that your YAML files do not contain mistakes. `yamllint` checks for valid syntax, common mistakes like key repetition and cosmetic problems such as line length, trailing spaces, wrong indentation, etc.

21.3 Any text file

A generic tool to check for common spelling mistakes is codespell.

CHAPTER

TWENTYTWO

DOCUMENTATION

The documentation lives on docs.esmvaltool.org.

22.1 Adding documentation

The documentation is built by readthedocs using Sphinx. There are two main ways of adding documentation:

1. As written text in the directory doc. When writing reStructuredText (`.rst`) files, please try to limit the line length to 80 characters and always start a sentence on a new line. This makes it easier to review changes to documentation on GitHub.
2. As docstrings or comments in code. For Python code, only the docstrings of Python modules, classes, and functions that are mentioned in doc/api are used to generate the online documentation. This results in the *ESMValCore API Reference*. The standard document with best practices on writing docstrings is PEP257. For the API documentation, we make use of numpy style docstrings.

22.2 What should be documented

Functionality that is visible to users should be documented. Any documentation that is visible on readthedocs should be well written and adhere to the standards for documentation. Examples of this include:

- The *recipe*
- Preprocessor *functions* and their *use from the recipe*
- *Configuration options*
- *Installation*
- *Output files*
- *Command line interface*
- *Diagnostic script interfaces*
- *The experimental Python interface*

Note that:

- For functions that compute scientific results, comments with references to papers and/or other resources as well as formula numbers should be included.
- When making changes to/introducing a new preprocessor function, also update the *preprocessor documentation*.

- There is no need to write complete numpy style documentation for functions that are not visible in the *ESMValCore API Reference* chapter on readthedocs. However, adding a docstring describing what a function does is always a good idea. For short functions, a one-line docstring is usually sufficient, but more complex functions might require slightly more extensive documentation.

When reviewing a pull request, always check that documentation is easy to understand and available in all expected places.

22.3 How to build and view the documentation

Whenever you make a pull request or push new commits to an existing pull request, readthedocs will automatically build the documentation. The link to the documentation will be shown in the list of checks below your pull request. Click 'Details' behind the check `docs/readthedocs.org:esmvalcore` to preview the documentation. If all checks were successful, you may need to click 'Show all checks' to see the individual checks.

To build the documentation on your own computer, go to the directory where the repository was cloned and run

```
sphinx-build doc doc/build
```

or

```
sphinx-build -Ea doc doc/build
```

to build it from scratch.

Make sure that your newly added documentation builds without warnings or errors and looks correctly formatted. CircleCI will build the documentation with the command:

```
sphinx-build -W doc doc/build
```

This will catch mistakes that can be detected automatically.

The configuration file for Sphinx is doc/shinx/source/conf.py.

See Integration with the ESMValCore documentation for information on how the ESMValCore documentation is integrated into the complete ESMValTool project documentation on readthedocs.

When reviewing a pull request, always check that the documentation checks shown below the pull request were successful.

CHAPTER

TWENTYTHREE

TESTS

To check that the code works correctly, there tests available in the tests directory. We use pytest to write and run our tests.

Contributions to ESMValCore should be covered by unit tests. Have a look at the existing tests in the `tests` directory for inspiration on how to write your own tests. If you do not know how to start with writing unit tests, ask the @ESMValGroup/tech-reviewers for help by commenting on the pull request and they will try to help you. It is also recommended that you have a look at the pytest documentation at some point when you start writing your own tests.

23.1 Running tests

To run the tests on your own computer, go to the directory where the repository is cloned and run the command

```
pytest
```

Optionally you can skip tests which require additional dependencies for supported diagnostic script languages by adding `-m 'not installation'` to the previous command. To only run tests from a single file, run the command

```
pytest tests/unit/test_some_file.py
```

If you would like to avoid loading the default pytest configuration from setup.cfg because this can be a bit slow for running just a few tests, use

```
pytest -c /dev/null tests/unit/test_some_file.py
```

Use

```
pytest --help
```

for more information on the available commands.

Whenever you make a pull request or push new commits to an existing pull request, the tests in the `tests` directory of the branch associated with the pull request will be run automatically on CircleCI. The results appear at the bottom of the pull request. Click on 'Details' for more information on a specific test job.

When reviewing a pull request, always check that all test jobs on CircleCI were successful.

23.2 Test coverage

To check which parts of your code are covered by unit tests, open the file `test-reports/coverage_html/index.html` (available after running a `pytest` command) and browse to the relevant file.

CircleCI will upload the coverage results from running the tests to codecov and Codacy. codecov is a service that will comment on pull requests with a summary of the test coverage. If codecov reports that the coverage has decreased, check the report and add additional tests. Alternatively, it is also possible to view code coverage on Codacy (click the Files tab) and CircleCI (open the `tests` job and click the ARTIFACTS tab). To see some of the results on CircleCI, Codacy, or codecov, you may need to log in; you can do so using your GitHub account.

When reviewing a pull request, always check that new code is covered by unit tests and codecov reports an increased coverage.

23.3 Sample data

New or modified preprocessor functions should preferably also be tested using the sample data. These tests are located in tests/sample_data. Please mark new tests that use the sample data with the decorator `@pytest.mark.use_sample_data`.

The ESMValTool_sample_data repository contains samples of CMIP6 data for testing ESMValCore. The ESMValTool-sample-data package is installed as part of the developer dependencies. The size of the package is relatively small (~ 100 MB), so it can be easily downloaded and distributed.

Preprocessing the sample data can be time-consuming, so some intermediate results are cached by pytest to make the tests run faster. If you suspect the tests are failing because the cache is invalid, clear it by running

```
pytest --cache-clear
```

To avoid running the time consuming tests that use sample data altogether, run

```
pytest -m "not use_sample_data"
```

23.4 Automated testing

Whenever you make a pull request or push new commits to an existing pull request, the tests in the `tests` of the branch associated with the pull request will be run automatically on CircleCI.

Every night, more extensive tests are run to make sure that problems with the installation of the tool are discovered by the development team before users encounter them. These nightly tests have been designed to follow the installation procedures described in the documentation, e.g. in the *Installation* chapter. The nightly tests are run using both CircleCI and GitHub Actions. The result of the tests ran by CircleCI can be seen on the CircleCI project page and the result of the tests ran by GitHub Actions can be viewed on the Actions tab of the repository (to learn more about the Github-hosted runners, please have a look the documentation).

The configuration of the tests run by CircleCI can be found in the directory .circleci, while the configuration of the tests run by GitHub Actions can be found in the directory .github/workflows.

CHAPTER

TWENTYFOUR

BACKWARD COMPATIBILITY

The ESMValCore package is used by many people to run their recipes. Many of these recipes are maintained in the public ESMValTool repository, but there are also users who choose not to share their work there. While our commitment is first and foremost to users who do share their recipes in the ESMValTool repository, we still try to be nice to all of the ESMValCore users.

Note: The backward compatibility policy outlines the key principles on backward compatibility and additional guidance on handling *backward-incompatible changes*. This policy applies to both, ESMValCore and ESMValTool.

When making changes, e.g. to the *recipe format*, the *diagnostic script interface*, the public *Python API*, or the *configuration file format*, keep in mind that this may affect many users. To keep the tool user friendly, try to avoid making changes that are not backward compatible, i.e. changes that require users to change their existing recipes, diagnostics, configuration files, or scripts.

If you really must change the public interfaces of the tool, always discuss this with the @ESMValGroup/esmvaltool-coreteam. Try to deprecate the feature first by issuing an *`ESMValCoreDeprecationWarning`* using the `warnings` module and schedule it for removal two minor versions from the upcoming release. For example, when you deprecate a feature in a pull request that will be included in version 2.5, that feature should be removed in version 2.7:

```
import warnings

from esmvalcore.exceptions import ESMValCoreDeprecationWarning

# Other code

def func(x, deprecated_option=None):
    """Deprecate deprecated_option."""
    if deprecated_option is not None:
        deprecation_msg = (
            "The option ``deprecated_option`` has been deprecated in "
            "ESMValCore version 2.5 and is scheduled for removal in "
            "version 2.7. Add additional text (e.g., description of "
            "alternatives) here.")
        warnings.warn(deprecation_msg, ESMValCoreDeprecationWarning)

    # Other code
```

Mention the version in which the feature will be removed in the deprecation message. Label the pull request with the deprecated feature label. When deprecating a feature, please follow up by actually removing the feature in due course.

If you must make backward incompatible changes, you need to update the available recipes in ESMValTool and link the ESMValTool pull request(s) in the ESMValCore pull request description. You can ask the @ESMValGroup/esmvaltool-

recipe-maintainers for help with updating existing recipes, but please be considerate of their time. You should tag the @ESMValGroup/esmvaltool-coreteam to notify them of the backward-incompatible change, and give at least 2 weeks for objections to be raised before merging to the main branch. If a strong objection is raised the backwards-incompatible change should not be merged until the objection is resolved.

When reviewing a pull request, always check for backward incompatible changes and make sure they are needed and have been discussed with the @ESMValGroup/esmvaltool-coreteam. Also, make sure the author of the pull request has created the accompanying pull request(s) to update the ESMValTool, before merging the ESMValCore pull request.

CHAPTER

TWENTYFIVE

DEPENDENCIES

Before considering adding a new dependency, carefully check that the license of the dependency you want to add and any of its dependencies are compatible with the Apache 2.0 license that applies to the ESMValCore. Note that GPL version 2 license is considered incompatible with the Apache 2.0 license, while the compatibility of GPL version 3 license with the Apache 2.0 license is questionable. See this statement by the authors of the Apache 2.0 license for more information.

When adding or removing dependencies, please consider applying the changes in the following files:

- `environment.yml` contains all the development dependencies; these are all from conda-forge
- `setup.py` contains all Python dependencies, regardless of their installation source

Note that packages may have a different name on conda-forge than on PyPI.

Several test jobs on CircleCI related to the installation of the tool will only run if you change the dependencies. These will be skipped for most pull requests.

When reviewing a pull request where dependencies are added or removed, always check that the changes have been applied in all relevant files.

CHAPTER

TWENTYSIX

LIST OF AUTHORS

If you make a contribution to ESMValCore and you would like to be listed as an author (e.g. on Zenodo), please add your name to the list of authors in `CITATION.cff` and generate the entry for the `.zenodo.json` file by running the commands

```
pip install cffconvert
cffconvert --ignore-suspect-keys --outputformat zenodo --outfile .zenodo.json
```

Presently, this method unfortunately discards entries *communities* and *grants* from that file; please restore them manually, or alternately proceed with the addition manually

CHAPTER

TWENTYSEVEN

PULL REQUEST CHECKS

To check that a pull request is up to standard, several automatic checks are run when you make a pull request. Read more about it in the *Tests* and *Documentation* sections. Successful checks have a green ✓ in front, a means the check failed.

If you need help with the checks, please ask the technical reviewer of your pull request for help. Ask @ESMValGroup/tech-reviewers if you do not have a technical reviewer yet.

If the checks are broken because of something unrelated to the current pull request, please check if there is an open issue that reports the problem. Create one if there is no issue yet. You can attract the attention of the @ESMValGroup/esmvaltool-coreteam by mentioning them in the issue if it looks like no-one is working on solving the problem yet. The issue needs to be fixed in a separate pull request first. After that has been merged into the `main` branch and all checks on this branch are green again, merge it into your own branch to get the tests to pass.

When reviewing a pull request, always make sure that all checks were successful. If the Codacy check keeps failing, please run prospector locally. If necessary, ask the pull request author to do the same and to address the reported issues. See the section on *code_quality* for more information. Never merge a pull request with failing CircleCI or readthedocs checks.

CHAPTER

TWENTYEIGHT

MAKING A RELEASE

The release manager makes the release, assisted by the release manager of the previous release, or if that person is not available, another previous release manager. Perform the steps listed below with two persons, to reduce the risk of error.

Note: The previous release manager ensures the current release manager has the required administrative permissions to make the release. Consider the following services: conda-forge, DockerHub, PyPI, and readthedocs.

The release of ESMValCore is tied to the release of ESMValTool. The detailed steps can be found in the ESMValTool documentation. To start the procedure, ESMValCore gets released as a release candidate to test the recipes in ESMValTool. If bugs are found during the testing phase of the release candidate, make as many release candidates for ESMValCore as needed in order to fix them.

To make a new release of the package, be it a release candidate or the final release, follow these steps:

28.1 1. Check that all tests and builds work

- Check that the `nightly` test run on CircleCI was successful.
- Check that the GitHub Actions test runs were successful.
- Check that the documentation builds successfully on readthedocs.
- Check that the Docker images are building successfully.

All tests should pass before making a release (branch).

28.2 2. Create a release branch

Create a branch off the `main` branch and push it to GitHub. Ask someone with administrative permissions to set up branch protection rules for it so only you and the person helping you with the release can push to it.

28.3 3. Increase the version number

The version number is automatically generated from the information provided by git using [setuptools-scm](https://pypi.org/project/setuptools-scm/), but a static version number is stored in `CITATION.cff`. Make sure to update the version number and release date in `CITATION.cff`. See https://semver.org for more information on choosing a version number. Make a pull request and get it merged into `main` and cherry pick it into the release branch.

28.4 4. Add release notes

Use the script esmvaltool/utils/draft_release_notes.py to create create a draft of the release notes. This script uses the titles and labels of merged pull requests since the previous release. Open a discussion to allow members of the development team to nominate pull requests as highlights. Add the most voted pull requests as highlights at the beginning of changelog. After the highlights section, list any backward incompatible changes that the release may include. The backward compatibility policy. lists the information that should be provided by the developer of any backward incompatible change. Make sure to also list any deprecations that the release may include, as well as a brief description on how to upgrade a deprecated feature. Review the results, and if anything needs changing, change it on GitHub and re-run the script until the changelog looks acceptable. Copy the result to the file `doc/changelog.rst`. Make a pull request and get it merged into `main` and cherry pick it into the release branch.

28.5 5. Make the (pre-)release on GitHub

Do a final check that all tests on CircleCI and GitHub Actions completed successfully. Then click the releases tab and create the new release from the release branch (i.e. not from `main`).

Create a tag and tick the *This is a pre-release* box if working with a release candidate.

28.6 6. Mark the release in the main branch

When the (pre-)release is tagged, it is time to merge the release branch back into *main*. We do this for two reasons, namely, one, to mark the point up to which commits in *main* have been considered for inclusion into the present release, and, two, to inform setuptools-scm about the version number so that it creates the correct version number in *main*. However, unlike in a normal merge, we do not want to integrate any of the changes from the release branch into main. This is because all changes that should be in both branches, i.e. bug fixes, originate from *main* anyway and the only other changes in the release branch relate to the release itself. To take this into account, we perform the merge in this case on the command line using the ours merge strategy (`git merge -s ours`), not to be confused with the `ours` option to the ort merge strategy (`git merge -X ours`). For details about merge strategies, see the above-linked page. To execute the merge use following sequence of steps

```
git fetch
git checkout main
git pull
git merge -s ours v2.1.x
git push
```

Note that the release branch remains intact and you should continue any work on the release on that branch.

28.7 7. Create and upload the PyPI package

The package is automatically uploaded to the PyPI by a GitHub action. If has failed for some reason, build and upload the package manually by following the instructions below.

Follow these steps to create a new Python package:

- Check out the tag corresponding to the release, e.g. `git checkout tags/v2.1.0`
- Make sure your current working directory is clean by checking the output of `git status` and by running `git clean -xdf` to remove any files ignored by git.
- Install the required packages: `python3 -m pip install --upgrade pep517 twine`
- Build the package: `python3 -m pep517.build --source --binary --out-dir dist/ .` This command should generate two files in the `dist` directory, e.g. `ESMValCore-2.3.1-py3-none-any.whl` and `ESMValCore-2.3.1.tar.gz`.
- Upload the package: `python3 -m twine upload dist/*` You will be prompted for an API token if you have not set this up before, see here for more information.

You can read more about this in Packaging Python Projects.

28.8 8. Create the Conda package

The `esmvalcore` package is published on the conda-forge conda channel. This is done via a pull request on the esmvalcore-feedstock repository.

To publish a release candidate, you have to open a pull request yourself. An example for this can be found here. Make sure to use the rc branch as the target branch for your pull request and follow all instructions given by the linter bot. The testing of ESMValTool will be performed with the published release candidate.

For the final release, this pull request is automatically opened by a bot. An example pull request can be found here. Follow the instructions by the bot to finalize the pull request. This step mostly contains updating dependencies that have been changed during the last release cycle. Once approved by the feedstock maintainers they will merge the pull request, which will in turn publish the package on conda-forge some time later. Contact the feedstock maintainers if you want to become a maintainer yourself.

28.9 9. Check the Docker images

There are two main Docker container images available for ESMValCore on Dockerhub:

- `esmvalgroup/esmvalcore:stable`, built from docker/Dockerfile, this is a tag that is always the same as the latest released version. This image is only built by Dockerhub when a new release is created.
- `esmvalgroup/esmvalcore:development`, built from docker/Dockerfile.dev, this is a tag that always contains the latest conda environment for ESMValCore, including any test dependencies. It is used by CircleCI to run the unit tests. This speeds up running the tests, as it avoids the need to build the conda environment for every test run. This image is built by Dockerhub every time there is a new commit to the `main` branch on Github.

In addition to the two images mentioned above, there is an image available for every release (e.g. `esmvalgroup/esmvalcore:v2.5.0`). When working on the Docker images, always try to follow the best practices.

After making the release, check that the Docker image for that release has been built correctly by

1. checking that the version tag is available on Dockerhub and the `stable` tag has been updated,

2. running some recipes with the `stable` tag Docker container, for example one recipe for Python, NCL, R, and Julia,
3. running a recipe with a Singularity container built from the `stable` tag.

If there is a problem with the automatically built container image, you can fix the problem and build a new image locally. For example, to build and upload the container image for v2.5.0 of the tool run:

```
git checkout v2.5.0
git clean -x
docker build -t esmvalgroup/esmvalcore:v2.5.0 . -f docker/Dockerfile
docker push esmvalgroup/esmvalcore:v2.5.0
```

(when making updates, you may want to add .post0, .post1, .. to the version number to avoid overwriting an older tag) and if it is the latest release that you are updating, also run

```
docker tag esmvalgroup/esmvalcore:v2.5.0 esmvalgroup/esmvalcore:stable
docker push esmvalgroup/esmvalcore:stable
```

Note that the `docker push` command will overwrite the existing tags on Dockerhub, but the previous container image will remain available as an untagged image.

Part VII

ESMValCore API Reference

ESMValCore is mostly used as a commandline tool. However, it is also possibly to use (parts of) ESMValTool as a library. This section documents the public API of ESMValCore.

CHAPTER

TWENTYNINE

CMOR FUNCTIONS

CMOR module.

29.1 Checking compliance

Module for checking iris cubes against their CMOR definitions.

Classes:

CheckLevels(value[, names, module, ...])	Level of strictness of the checks.
CMORCheck(cube, var_info[, frequency, ...])	Class used to check the CMOR-compliance of the data.

Exceptions:

CMORCheckError	Exception raised when a cube does not pass the CMORCheck.

Functions:

cmor_check_metadata(cube, cmor_table, mip, ...)	Check if metadata conforms to variable's CMOR definition.
cmor_check_data(cube, cmor_table, mip, ...)	Check if data conforms to variable's CMOR definition.
cmor_check(cube, cmor_table, mip, short_name)	Check if cube conforms to variable's CMOR definition.

class esmvalcore.cmor.check.**CheckLevels**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Level of strictness of the checks.

Attributes:

DEBUG	Report any debug message that the checker wants to communicate.
STRICT	Fail if there are warnings regarding compliance of CMOR standards.
DEFAULT	Fail if cubes present any discrepancy with CMOR standards.
RELAXED	Fail if cubes present severe discrepancies with CMOR standards.
IGNORE	Do not fail for any discrepancy with CMOR standards.

DEBUG = 1

Report any debug message that the checker wants to communicate.

STRICT = 2

Fail if there are warnings regarding compliance of CMOR standards.

DEFAULT = 3

Fail if cubes present any discrepancy with CMOR standards.

RELAXED = 4

Fail if cubes present severe discrepancies with CMOR standards.

IGNORE = 5

Do not fail for any discrepancy with CMOR standards.

exception esmvalcore.cmor.check.**CMORCheckError**

Bases: Exception

Exception raised when a cube does not pass the CMORCheck.

class esmvalcore.cmor.check.**CMORCheck**(*cube*, *var_info*, *frequency=None*, *fail_on_error=False*, *check_level=CheckLevels.DEFAULT*, *automatic_fixes=False*)

Bases: object

Class used to check the CMOR-compliance of the data.

Parameters

- **cube** (*iris.cube.Cube:*) – Iris cube to check.
- **var_info** (*variables_info.VariableInfo*) – Variable info to check.
- **frequency** (*str*) – Expected frequency for the data. If not given, use the one from the variable information.
- **fail_on_error** (*bool*) – If true, CMORCheck stops on the first error. If false, it collects all possible errors before stopping.
- **automatic_fixes** (*bool*) – If True, CMORCheck will try to apply automatic fixes for any detected error, if possible.

  Deprecated since version 2.10.0: This option has been deprecated in ESMValCore version 2.10.0 and is scheduled for removal in version 2.12.0. Please use the functions *fix_metadata()*, *fix_data()*, or *esmvalcore.dataset.Dataset.load()* (which automatically includes the first two functions) instead. Fixes and CMOR checks have been clearly separated in ESMValCore version 2.10.0.
- **check_level** (CheckLevels) – Level of strictness of the checks.

frequency

Expected frequency for the data.

Type
str

Methods:

check_metadata([logger])	Check the cube metadata.
check_data([logger])	Check the cube data.
report_errors()	Report detected errors.
report_warnings()	Report detected warnings to the given logger.
report_debug_messages()	Report detected debug messages to the given logger.
has_errors()	Check if there are reported errors.
has_warnings()	Check if there are reported warnings.
has_debug_messages()	Check if there are reported debug messages.
report(level, message, *args)	Report a message from the checker.
report_critical(message, *args)	Report an error.
report_error(message, *args)	Report a normal error.
report_warning(message, *args)	Report a warning level error.
report_debug_message(message, *args)	Report a debug message.

check_metadata(*logger: Logger | None = None*) → Cube

Check the cube metadata.

It will also report some warnings in case of minor errors.

Parameters
logger (*Logger | None*) – Given logger.

Returns
Checked cube.

Return type
iris.cube.Cube

Raises
CMORCheckError – If errors are found. If fail_on_error attribute is set to True, raises as soon as an error is detected. If set to False, it perform all checks and then raises.

check_data(*logger: Logger | None = None*) → Cube

Check the cube data.

Assumes that metadata is correct, so you must call check_metadata prior to this.

It will also report some warnings in case of minor errors.

Parameters
logger (*Logger | None*) – Given logger.

Returns
Checked cube.

Return type
iris.cube.Cube

Raises
CMORCheckError – If errors are found. If fail_on_error attribute is set to True, raises as soon as an error is detected. If set to False, it perform all checks and then raises.

`report_errors`()

Report detected errors.

Raises
CMORCheckError – If any errors were reported before calling this method.

`report_warnings`()

Report detected warnings to the given logger.

`report_debug_messages`()

Report detected debug messages to the given logger.

`has_errors`()

Check if there are reported errors.

Returns
True if there are pending errors, False otherwise.

Return type
bool

`has_warnings`()

Check if there are reported warnings.

Returns
True if there are pending warnings, False otherwise.

Return type
bool

`has_debug_messages`()

Check if there are reported debug messages.

Returns
True if there are pending debug messages, False otherwise.

Return type
bool

`report`(*level*, *message*, **args*)

Report a message from the checker.

Parameters

- **level** (CheckLevels) – Message level
- **message** (*str*) – Message to report
- **args** – String format args for the message

Raises
CMORCheckError – If fail on error is set, it is thrown when registering an error message

`report_critical`(*message*, **args*)

Report an error.

If fail_on_error is set to True, raises automatically. If fail_on_error is set to False, stores it for later reports.

Parameters

- **message** (*str: unicode*) – Message for the error.
- ***args** – arguments to format the message string.

report_error(*message*, **args*)

Report a normal error.

Parameters

- **message** (*str: unicode*) – Message for the error.
- ***args** – arguments to format the message string.

report_warning(*message*, **args*)

Report a warning level error.

Parameters

- **message** (*str: unicode*) – Message for the warning.
- ***args** – arguments to format the message string.

report_debug_message(*message*, **args*)

Report a debug message.

Parameters

- **message** (*str: unicode*) – Message for the debug logger.
- ***args** – arguments to format the message string

esmvalcore.cmor.check.**cmor_check_metadata**(*cube: Cube*, *cmor_table: str*, *mip: str*, *short_name: str*, *frequency: str | None = None*, *check_level:* CheckLevels *= CheckLevels.DEFAULT*) → Cube

Check if metadata conforms to variable's CMOR definition.

None of the checks at this step will force the cube to load the data.

Parameters

- **cube** (*Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use (i.e., the variable's project).
- **mip** (*str*) – Variable's MIP.
- **short_name** (*str*) – Variable's short name.
- **frequency** (*str | None*) – Data frequency. If not given, use the one from the CMOR table of the variable.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Returns
Checked cube.

Return type
iris.cube.Cube

esmvalcore.cmor.check.**cmor_check_data**(*cube: Cube*, *cmor_table: str*, *mip: str*, *short_name: str*, *frequency: str | None = None*, *check_level:* CheckLevels *= CheckLevels.DEFAULT*) → Cube

Check if data conforms to variable's CMOR definition.

Parameters

- **cube** (*Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use (i.e., the variable's project).

- **mip** (*str*) – Variable's MIP.
- **short_name** (*str*) – Variable's short name
- **frequency** (*str | None*) – Data frequency. If not given, use the one from the CMOR table of the variable.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Returns
Checked cube.

Return type
iris.cube.Cube

esmvalcore.cmor.check.**cmor_check**(*cube: Cube*, *cmor_table: str*, *mip: str*, *short_name: str*, *frequency: str | None = None*, *check_level: CheckLevels = CheckLevels.DEFAULT*) → Cube

Check if cube conforms to variable's CMOR definition.

Equivalent to calling *cmor_check_metadata()* and *cmor_check_data()* consecutively.

Parameters

- **cube** (*Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use (i.e., the variable's project).
- **mip** (*str*) – Variable's MIP.
- **short_name** (*str*) – Variable's short name.
- **frequency** (*str | None*) – Data frequency. If not given, use the one from the CMOR table of the variable.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Returns
Checked cube.

Return type
iris.cube.Cube

29.2 Automatically fixing issues

Apply automatic fixes for known errors in cmorized data.

All functions in this module will work even if no fixes are available for the given dataset. Therefore is recommended to apply them to all variables to be sure that all known errors are fixed.

Functions:

fix_data(cube, short_name, project, dataset, mip)	Fix cube data if fixes are required.
fix_file(file, short_name, project, dataset, ...)	Fix files before ESMValTool can load them.
fix_metadata(cubes, short_name, project, ...)	Fix cube metadata if fixes are required.

esmvalcore.cmor.fix.**fix_data**(*cube: Cube*, *short_name: str*, *project: str*, *dataset: str*, *mip: str*, *frequency: str | None = None*, *check_level: CheckLevels = CheckLevels.DEFAULT*, *session: Session | None = None*, ***extra_facets*) → Cube

Fix cube data if fixes are required.

This method assumes that metadata is already fixed and checked.

This method collects all the relevant fixes (including generic ones) for a given variable and applies them.

Parameters

- **cube** (*Cube*) – Cube to fix.
- **short_name** (*str*) – Variable's short name.
- **project** (*str*) – Project of the dataset.
- **dataset** (*str*) – Name of the dataset.
- **mip** (*str*) – Variable's MIP.
- **frequency** (*Optional[str]*) – Variable's data frequency, if available.
- **check_level** (CheckLevels) – Level of strictness of the checks.

  Deprecated since version 2.10.0: This option has been deprecated in ESMValCore version 2.10.0 and is scheduled for removal in version 2.12.0. Please use the functions *cmor_check_metadata()*, *cmor_check_data()*, or *cmor_check()* instead. This function will no longer perform CMOR checks. Fixes and CMOR checks have been clearly separated in ESMValCore version 2.10.0.
- **session** (*Optional[Session]*) – Current session which includes configuration and directory information.
- ****extra_facets** – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns
Fixed cube.

Return type
iris.cube.Cube

esmvalcore.cmor.fix.**fix_file**(*file: Path*, *short_name: str*, *project: str*, *dataset: str*, *mip: str*, *output_dir: Path*, *add_unique_suffix: bool = False*, *session:* Session | *None = None*, *frequency: str | None = None*, ***extra_facets*) → str | Path

Fix files before ESMValTool can load them.

These fixes are only for issues that prevent iris from loading the cube or that cannot be fixed after the cube is loaded.

Original files are not overwritten.

Parameters

- **file** (*Path*) – Path to the original file.
- **short_name** (*str*) – Variable's short name.
- **project** (*str*) – Project of the dataset.
- **dataset** (*str*) – Name of the dataset.
- **mip** (*str*) – Variable's MIP.
- **output_dir** (*Path*) – Output directory for fixed files.
- **add_unique_suffix** (*bool*) – Adds a unique suffix to *output_dir* for thread safety.

- **session** (*Optional[Session]*) – Current session which includes configuration and directory information.
- **frequency** (*Optional[str]*) – Variable's data frequency, if available.
- ****extra_facets** – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns
Path to the fixed file.

Return type
str or pathlib.Path

esmvalcore.cmor.fix.**fix_metadata**(*cubes: Sequence[Cube]*, *short_name: str*, *project: str*, *dataset: str*, *mip: str*, *frequency: str | None = None*, *check_level:* CheckLevels *= CheckLevels.DEFAULT*, *session:* Session *| None = None*, ***extra_facets*) → CubeList

Fix cube metadata if fixes are required.

This method collects all the relevant fixes (including generic ones) for a given variable and applies them.

Parameters

- **cubes** (*Sequence[Cube]*) – Cubes to fix.
- **short_name** (*str*) – Variable's short name.
- **project** (*str*) – Project of the dataset.
- **dataset** (*str*) – Name of the dataset.
- **mip** (*str*) – Variable's MIP.
- **frequency** (*Optional[str]*) – Variable's data frequency, if available.
- **check_level** (CheckLevels) – Level of strictness of the checks.

  Deprecated since version 2.10.0: This option has been deprecated in ESMValCore version 2.10.0 and is scheduled for removal in version 2.12.0. Please use the functions *cmor_check_metadata()*, *cmor_check_data()*, or *cmor_check()* instead. This function will no longer perform CMOR checks. Fixes and CMOR checks have been clearly separated in ESMValCore version 2.10.0.
- **session** (*Optional[Session]*) – Current session which includes configuration and directory information.
- ****extra_facets** – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns
Fixed cubes.

Return type
iris.cube.CubeList

29.3 Functions for fixing issues

Functions for fixing specific issues with datasets.

Functions:

add_altitude_from_plev(cube)	Add altitude coordinate from pressure level coordinate.
add_plev_from_altitude(cube)	Add pressure level coordinate from altitude coordinate.
get_next_month(month, year)	Get next month and year.
get_time_bounds(time, freq)	Get bounds for time coordinate.

esmvalcore.cmor.fixes.**add_altitude_from_plev**(*cube*)

Add altitude coordinate from pressure level coordinate.

Parameters
cube (*iris.cube.Cube*) – Input cube.

Raises
ValueError – cube does not contain coordinate `air_pressure`.

esmvalcore.cmor.fixes.**add_plev_from_altitude**(*cube*)

Add pressure level coordinate from altitude coordinate.

Parameters
cube (*iris.cube.Cube*) – Input cube.

Raises
ValueError – cube does not contain coordinate `altitude`.

esmvalcore.cmor.fixes.**get_next_month**(*month: int*, *year: int*) → tuple[int, int]

Get next month and year.

Parameters

- **month** (*int*) – Current month.
- **year** (*int*) – Current year.

Returns
Next month and next year.

Return type
tuple[int, int]

esmvalcore.cmor.fixes.**get_time_bounds**(*time: Coord*, *freq: str*) → ndarray

Get bounds for time coordinate.

For monthly data, use the first day of the current month and the first day of the next month. For yearly or decadal data, use 1 January of the current year and 1 January of the next year or 10 years from the current year. For other frequencies (daily, 6-hourly, 3-hourly, hourly), half of the frequency is subtracted/added from the current point in time to get the bounds.

Parameters

- **time** (*Coord*) – Time coordinate.
- **freq** (*str*) – Frequency.

Returns
Time bounds

Return type
np.ndarray

Raises
`NotImplementedError` – Non-supported frequency is given.

29.4 Using CMOR tables

CMOR information reader for ESMValTool.

Read variable information from CMOR 2 and CMOR 3 tables and make it easily available for the other components of ESMValTool

Classes:

CMIP3Info(cmor_tables_path[, default, ...])	Class to read CMIP3-like data request.
CMIP5Info(cmor_tables_path[, default, ...])	Class to read CMIP5-like data request.
CMIP6Info(cmor_tables_path[, default, ...])	Class to read CMIP6-like data request.
CoordinateInfo(name)	Class to read and store coordinate information.
CustomInfo([cmor_tables_path])	Class to read custom var info for ESMVal.
InfoBase(default, alt_names, strict)	Base class for all table info classes.
JsonInfo()	Base class for the info classes.
TableInfo(*args, **kwargs)	Container class for storing a CMOR table.
VariableInfo(table_type, short_name)	Class to read and store variable information.

Data:

CMOR_TABLES	CMOR info objects.

Functions:

get_var_info(project, mip, short_name)	Get variable information.
read_cmor_tables([cfg_developer])	Read cmor tables required in the configuration.

class esmvalcore.cmor.table.**CMIP3Info**(*cmor_tables_path*, *default=None*, *alt_names=None*, *strict=True*)

Bases: *CMIP5Info*

Class to read CMIP3-like data request.

Parameters

- **cmor_tables_path** (*str*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable information.

get_table(*table*)

Search and return the table info.

Parameters
table (*str*) – Table name

Returns
Return the TableInfo object for the requested table if found, returns None if not

Return type
TableInfo

get_variable(*table_name: str*, *short_name: str*, *derived: bool | None = False*) → *VariableInfo* | None

Search and return the variable information.

Parameters

- **table_name** (*str*) – Table name, i.e., the variable's MIP.
- **short_name** (*str*) – Variable's short name.
- **derived** (*bool | None*) – Variable is derived. Information retrieval for derived variables always looks in the default tables (usually, the custom tables) if variable is not found in the requested table.

Returns
VariableInfo object for the requested variable if found, `None` otherwise.

Return type
VariableInfo | None

class esmvalcore.cmor.table.**CMIP5Info**(*cmor_tables_path*, *default=None*, *alt_names=None*, *strict=True*)

Bases: *InfoBase*

Class to read CMIP5-like data request.

Parameters

- **cmor_tables_path** (*str*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable information.

get_table(*table*)

Search and return the table info.

Parameters
table (*str*) – Table name

Returns
Return the TableInfo object for the requested table if found, returns None if not

Return type
TableInfo

get_variable(*table_name: str*, *short_name: str*, *derived: bool | None = False*) → *VariableInfo* | None

Search and return the variable information.

Parameters

- **table_name** (*str*) – Table name, i.e., the variable's MIP.
- **short_name** (*str*) – Variable's short name.
- **derived** (*bool | None*) – Variable is derived. Information retrieval for derived variables always looks in the default tables (usually, the custom tables) if variable is not found in the requested table.

Returns

VariableInfo object for the requested variable if found, None otherwise.

Return type

VariableInfo | None

class esmvalcore.cmor.table.**CMIP6Info**(*cmor_tables_path*, *default=None*, *alt_names=None*, *strict=True*, *default_table_prefix=''*)

Bases: *InfoBase*

Class to read CMIP6-like data request.

This uses CMOR 3 json format

Parameters

- **cmor_tables_path** (*str*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable information.

get_table(*table*)

Search and return the table info.

Parameters

table (*str*) – Table name

Returns

Return the TableInfo object for the requested table if found, returns None if not

Return type

TableInfo

get_variable(*table_name: str*, *short_name: str*, *derived: bool | None = False*) → *VariableInfo* | None

Search and return the variable information.

Parameters

- **table_name** (*str*) – Table name, i.e., the variable's MIP.
- **short_name** (*str*) – Variable's short name.

- **derived** (*bool | None*) – Variable is derived. Information retrieval for derived variables always looks in the default tables (usually, the custom tables) if variable is not found in the requested table.

Returns
VariableInfo object for the requested variable if found, None otherwise.

Return type
VariableInfo | None

esmvalcore.cmor.table.**CMOR_TABLES**: dict[str, *CMIP3Info* | *CMIP5Info* | *CMIP6Info* | *CustomInfo*] = {'CESM': <esmvalcore.cmor.table.CMIP6Info object>, 'CMIP3': <esmvalcore.cmor.table.CMIP3Info object>, 'CMIP5': <esmvalcore.cmor.table.CMIP5Info object>, 'CMIP6': <esmvalcore.cmor.table.CMIP6Info object>, 'CORDEX': <esmvalcore.cmor.table.CMIP5Info object>, 'EMAC': <esmvalcore.cmor.table.CMIP6Info object>, 'ICON': <esmvalcore.cmor.table.CMIP6Info object>, 'IPSLCM': <esmvalcore.cmor.table.CMIP6Info object>, 'OBS': <esmvalcore.cmor.table.CMIP5Info object>, 'OBS6': <esmvalcore.cmor.table.CMIP6Info object>, 'ana4mips': <esmvalcore.cmor.table.CMIP5Info object>, 'custom': <esmvalcore.cmor.table.CustomInfo object>, 'native6': <esmvalcore.cmor.table.CMIP6Info object>, 'obs4MIPs': <esmvalcore.cmor.table.CMIP6Info object>}

CMOR info objects.

Type
dict of str, obj

class esmvalcore.cmor.table.**CoordinateInfo**(*name*)

Bases: *JsonInfo*

Class to read and store coordinate information.

Attributes:

axis	Axis
generic_lev_name	Generic level name
long_name	Long name
must_have_bounds	Whether bounds are required on this dimension
out_name	Out name
requested	Values requested
standard_name	Standard name
stored_direction	Direction in which the coordinate increases
units	Units
valid_max	Maximum allowed value
valid_min	Minimum allowed value
value	Coordinate value
var_name	Short name

Methods:

read_json(json_data)	Read coordinate information from json.

axis

Axis

generic_lev_name
Generic level name

long_name
Long name

must_have_bounds
Whether bounds are required on this dimension

out_name
Out name

This is the name of the variable in the file

read_json(*json_data*)
Read coordinate information from json.

Non-present options will be set to empty

Parameters
json_data (*dict*) – dictionary created by the json reader containing coordinate information

requested
Values requested

standard_name
Standard name

stored_direction
Direction in which the coordinate increases

units
Units

valid_max
Maximum allowed value

valid_min
Minimum allowed value

value
Coordinate value

var_name
Short name

class esmvalcore.cmor.table.**CustomInfo**(*cmor_tables_path=None*)
Bases: *CMIP5Info*

Class to read custom var info for ESMVal.

Parameters
cmor_tables_path (*str or None*) – Full path to the table or name for the table if it is present in ESMValTool repository

Methods:

get_table(table)	Search and return the table info.
get_variable(table, short_name[, derived])	Search and return the variable info.

get_table(*table*)

Search and return the table info.

Parameters
table (*str*) – Table name

Returns
Return the TableInfo object for the requested table if found, returns None if not

Return type
TableInfo

get_variable(*table*, *short_name*, *derived=False*)

Search and return the variable info.

Parameters

- **table** (*str*) – Table name
- **short_name** (*str*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval for derived variables always look on the default tables if variable is not find in the requested table

Returns
Return the VariableInfo object for the requested variable if found, returns None if not

Return type
VariableInfo

class esmvalcore.cmor.table.**InfoBase**(*default*, *alt_names*, *strict*)

Bases: object

Base class for all table info classes.

This uses CMOR 3 json format

Parameters

- **default** (*object*) – Default table to look variables on if not found
- **alt_names** (*list[list[str]]*) – List of known alternative names for variables
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable information.

get_table(*table*)

Search and return the table info.

Parameters
table (*str*) – Table name

Returns
Return the TableInfo object for the requested table if found, returns None if not

Return type
TableInfo

get_variable(*table_name: str*, *short_name: str*, *derived: bool | None = False*) → *VariableInfo* | None

Search and return the variable information.

Parameters

- **table_name** (*str*) – Table name, i.e., the variable's MIP.
- **short_name** (*str*) – Variable's short name.
- **derived** (*bool | None*) – Variable is derived. Information retrieval for derived variables always looks in the default tables (usually, the custom tables) if variable is not found in the requested table.

Returns

VariableInfo object for the requested variable if found, None otherwise.

Return type

VariableInfo | None

class esmvalcore.cmor.table.**JsonInfo**

Bases: object

Base class for the info classes.

Provides common utility methods to read json variables

class esmvalcore.cmor.table.**TableInfo**(**args*, ***kwargs*)

Bases: dict

Container class for storing a CMOR table.

Methods:

clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(*value=None*, */*)
Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()
Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class esmvalcore.cmor.table.**VariableInfo**(*table_type*, *short_name*)
Bases: *JsonInfo*

Class to read and store variable information.

Attributes:

coordinates	Coordinates
dimensions	List of dimensions
frequency	Data frequency
long_name	Long name
modeling_realm	Modeling realm
positive	Increasing direction
short_name	Short name
standard_name	Standard name
units	Data units
valid_max	Maximum admitted value
valid_min	Minimum admitted value

Methods:

copy()	Return a shallow copy of VariableInfo.
has_coord_with_standard_name(standard_name)	Check if a coordinate with a given *standard_name* exists.
read_json(json_data, default_freq)	Read variable information from json.

coordinates

Coordinates

This is a dict with the names of the dimensions as keys and CoordinateInfo objects as values.

copy()

Return a shallow copy of VariableInfo.

Returns
Shallow copy of this object.

Return type
VariableInfo

dimensions

List of dimensions

frequency

Data frequency

has_coord_with_standard_name(*standard_name: str*) → bool

Check if a coordinate with a given *standard_name* exists.

For some coordinates, multiple (slightly different) versions with different dimension names but identical *standard_name* exist. For example, the CMIP6 tables provide 4 different *standard_name=time* dimensions: *time*, *time1*, *time2*, and *time3*. Other examples would be the CMIP6 pressure levels (*plev19*, *plev23*, *plev27*, etc. with standard name *air_pressure*) and the altitudes (*alt16*, *alt40* with standard name *altitude*).

This function can be used to check for the existence of a specific coordinate defined by its *standard_name*, not its dimension name.

Parameters
standard_name (*str*) – Standard name to be checked.

Returns
True if there is at least one coordinate with the given *standard_name*, *False* if not.

Return type
bool

long_name

Long name

modeling_realm

Modeling realm

positive

Increasing direction

read_json(*json_data*, *default_freq*)

Read variable information from json.

Non-present options will be set to empty

Parameters

- **json_data** (*dict*) – Dictionary created by the json reader containing variable information.
- **default_freq** (*str*) – Default frequency to use if it is not defined at variable level.

short_name

Short name

standard_name
Standard name

units
Data units

valid_max
Maximum admitted value

valid_min
Minimum admitted value

esmvalcore.cmor.table.**get_var_info**(*project: str*, *mip: str*, *short_name: str*) → *VariableInfo* | None

Get variable information.

Note: If *project=CORDEX* and the *mip* ends with 'hr', it is cropped to 'h' since CORDEX X-hourly tables define the *mip* as ending in 'h' instead of 'hr'.

Parameters

- **project** (*str*) – Dataset's project.
- **mip** (*str*) – Variable's CMOR table, i.e., MIP.
- **short_name** (*str*) – Variable's short name.

Returns
VariableInfo object for the requested variable if found, `None` otherwise.

Return type
VariableInfo | None

Raises
KeyError – No CMOR tables available for *project*.

esmvalcore.cmor.table.**read_cmor_tables**(*cfg_developer: Path | None = None*) → None

Read cmor tables required in the configuration.

Parameters
cfg_developer (*Path | None*) – Path to config-developer.yml file.

Raises
TypeError – If *cfg_developer* is not a Path-like object

Return type
None

CHAPTER

THIRTY

CONFIGURATION

This section describes the *config* module.

30.1 Config

Configuration of ESMValCore/Tool is done via the *Config* object. The global configuration can be imported from the *esmvalcore.config* module as *CFG*:

```
>>> from esmvalcore.config import CFG
>>> CFG
Config({'auxiliary_data_dir': PosixPath('/home/user/auxiliary_data'),
        'compress_netcdf': False,
        'config_developer_file': None,
        'config_file': PosixPath('/home/user/.esmvaltool/config-user.yml'),
        'drs': {'CMIP5': 'default', 'CMIP6': 'default'},
        'exit_on_warning': False,
        'log_level': 'info',
        'max_parallel_tasks': None,
        'output_dir': PosixPath('/home/user/esmvaltool_output'),
        'output_file_type': 'png',
        'profile_diagnostic': False,
        'remove_preproc_dir': True,
        'rootpath': {'CMIP5': '~/default_inputpath',
                     'CMIP6': '~/default_inputpath',
                     'default': '~/default_inputpath'},
        'save_intermediary_cubes': False)
```

The parameters for the user configuration file are listed *here*.

CFG is essentially a python dictionary with a few extra functions, similar to `matplotlib.rcParams`. This means that values can be updated like this:

```
>>> CFG['output_dir'] = '~/esmvaltool_output'
>>> CFG['output_dir']
PosixPath('/home/user/esmvaltool_output')
```

Notice that *CFG* automatically converts the path to an instance of `pathlib.Path` and expands the home directory. All values entered into the config are validated to prevent mistakes, for example, it will warn you if you make a typo in the key:

```
>>> CFG['output_directory'] = '~/esmvaltool_output'
InvalidConfigParameter: `output_directory` is not a valid config parameter.
```

Or, if the value entered cannot be converted to the expected type:

```
>>> CFG['max_parallel_tasks'] = ''
InvalidConfigParameter: Key `max_parallel_tasks`: Could not convert '' to int
```

Config is also flexible, so it tries to correct the type of your input if possible:

```
>>> CFG['max_parallel_tasks'] = '8'  # str
>>> type(CFG['max_parallel_tasks'])
int
```

By default, the config is loaded from the default location (`/home/user/.esmvaltool/config-user.yml`). If it does not exist, it falls back to the default values. to load a different file:

```
>>> CFG.load_from_file('~/my-config.yml')
```

Or to reload the current config:

```
>>> CFG.reload()
```

30.2 Session

Recipes and diagnostics will be run in their own directories. This behaviour can be controlled via the *Session* object. A *Session* can be initiated from the global *Config*.

```
>>> session = CFG.start_session(name='my_session')
```

A *Session* is very similar to the config. It is also a dictionary, and copies all the keys from the *Config*. At this moment, `session` is essentially a copy of *CFG*:

```
>>> print(session == CFG)
True
>>> session['output_dir'] = '~/my_output_dir'
>>> print(session == CFG)  # False
False
```

A *Session* also knows about the directories where the data will stored. The session name is used to prefix the directories.

```
>>> session.session_dir
/home/user/my_output_dir/my_session_20201203_155821
>>> session.run_dir
/home/user/my_output_dir/my_session_20201203_155821/run
>>> session.work_dir
/home/user/my_output_dir/my_session_20201203_155821/work
>>> session.preproc_dir
/home/user/my_output_dir/my_session_20201203_155821/preproc
>>> session.plot_dir
/home/user/my_output_dir/my_session_20201203_155821/plots
```

Unlike the global configuration, of which only one can exist, multiple sessions can be initiated from *Config*.

30.3 API reference

Configuration module.

esmvalcore.config.**CFG**

ESMValCore configuration.

By default this will be loaded from the file `~/.esmvaltool/config-user.yml`.

Classes:

Config(*args, **kwargs)	ESMValTool configuration object.
Session(config[, name])	Container class for session configuration and directory information.

class esmvalcore.config.**Config**(**args*, ***kwargs*)

ESMValTool configuration object.

Do not instantiate this class directly, but use *esmvalcore.config.CFG* instead.

Methods:

load_from_file([filename])	Load user configuration from the given file.
reload()	Reload the config file.
start_session(name)	Start a new session from this configuration object.

load_from_file(*filename: PathLike | str | None = None*) → None

Load user configuration from the given file.

Parameters
filename (*PathLike | str | None*) –

Return type
None

reload()

Reload the config file.

start_session(*name: str*)

Start a new session from this configuration object.

Parameters
name (*str*) – Name of the session.

Return type
Session

class esmvalcore.config.**Session**(*config: dict*, *name: str = 'session'*)

Container class for session configuration and directory information.

Do not instantiate this class directly, but use `CFG.start_session` instead.

Parameters

- **config** (*dict*) – Dictionary with configuration settings.
- **name** (*str*) – Name of the session to initialize, for example, the name of the recipe (default='session').

Attributes:

config_dir	Return user config directory.
main_log	Return main log file.
main_log_debug	Return main log debug file.
plot_dir	Return plot directory.
preproc_dir	Return preproc directory.
relative_main_log	
relative_main_log_debug	
relative_plot_dir	
relative_preproc_dir	
relative_run_dir	
relative_work_dir	
run_dir	Return run directory.
session_dir	Return session directory.
work_dir	Return work directory.

Methods:

set_session_name([name])	Set the name for the session.

property config_dir

Return user config directory.

property main_log

Return main log file.

property main_log_debug

Return main log debug file.

property plot_dir

Return plot directory.

property preproc_dir

Return preproc directory.

relative_main_log = PosixPath('run/main_log.txt')

relative_main_log_debug = PosixPath('run/main_log_debug.txt')

relative_plot_dir = PosixPath('plots')

relative_preproc_dir = PosixPath('preproc')

relative_run_dir = PosixPath('run')

relative_work_dir = PosixPath('work')

property run_dir

Return run directory.

property session_dir

Return session directory.

session_name: str | None

set_session_name(*name: str = 'session'*)

Set the name for the session.

The *name* is used to name the session directory, e.g. *session_20201208_132800/*. The date is suffixed automatically.

Parameters
name (*str*) –

property work_dir

Return work directory.

CHAPTER

THIRTYONE

DATASET

Classes and functions for defining, finding, and loading data.

Classes:

Dataset(**facets)	Define datasets, find the related files, and load them.

Data:

INHERITED_FACETS	Inherited facets.

Functions:

datasets_to_recipe(datasets[, recipe])	Create or update a recipe from datasets.

class esmvalcore.dataset.**Dataset**(***facets: str | Sequence[str] | Number*)

Define datasets, find the related files, and load them.

Parameters

****facets** (*FacetValue*) – Facets describing the dataset. See *esmvalcore.esgf.facets.FACETS* for the mapping between the facet names used by ESMValCore and those used on ESGF.

supplementaries

List of supplementary datasets.

Type

list[*Dataset*]

facets

Facets describing the dataset.

Type

esmvalcore.typing.Facets

Methods:

add_supplementary(**facets)	Add an supplementary dataset.
augment_facets()	Add extra facets.
copy(**facets)	Create a copy.
find_files()	Find files.
from_files()	Create datasets based on the available files.
from_ranges()	Create a list of datasets from short notations.
from_recipe(recipe, session)	Read datasets from a recipe.
load()	Load dataset.
set_facet(key, value[, persist])	Set facet.
set_version()	Set the `'version'` facet based on the available data.
summary([shorten])	Summarize the content of dataset.

Attributes:

files	The files associated with this dataset.
minimal_facets	Return a dictionary with the persistent facets.
session	A *esmvalcore.config.Session* associated with the dataset.

add_supplementary(***facets: str | Sequence[str] | Number*) → None

Add an supplementary dataset.

This is a convenience function that will create a copy of the current dataset, update its facets with the values specified in `**facets`, and append it to *Dataset.supplementaries*. For more control over the creation of the supplementary dataset, first create a new *Dataset* describing the supplementary dataset and then append it to *Dataset.supplementaries*.

Parameters
****facets** (*str | Sequence[str] | Number*) – Facets describing the supplementary variable.

Return type
None

augment_facets() → None

Add extra facets.

This function will update the dataset with additional facets from various sources.

Return type
None

copy(***facets: str | Sequence[str] | Number*) → *Dataset*

Create a copy.

Parameters
****facets** (*str | Sequence[str] | Number*) – Update these facets in the copy. Note that for supplementary datasets attached to the dataset, the `'short_name'` and `'mip'` facets will not be updated with these values.

Returns
A copy of the dataset.

Return type
Dataset

property files: Sequence[*ESGFFile* | *LocalFile*]

The files associated with this dataset.

find_files() → None

Find files.

Look for files and populate the *Dataset.files* property of the dataset and its supplementary datasets.

Return type
None

from_files() → Iterator[*Dataset*]

Create datasets based on the available files.

The facet values for local files are retrieved from the directory tree where the directories represent the facets values. Reading facet values from file names is not yet supported. See *CMIP data* for more information on this kind of file organization.

`glob.glob()` patterns can be used as facet values to select multiple datasets. If for some of the datasets not all glob patterns can be expanded (e.g. because the required facet values cannot be inferred from the directory names), these datasets will be ignored, unless this happens to be all datasets.

If `glob.glob()` patterns are used in supplementary variables and multiple matching datasets are found, only the supplementary dataset that has most facets in common with the main dataset will be attached.

Supplementary datasets will in inherit the facet values from the main dataset for those facets listed in *INHERITED_FACETS*.

Examples

See *Discovering data* for example use cases.

Yields
Dataset – Datasets representing the available files.

Return type
Iterator[Dataset]

from_ranges() → list[*Dataset*]

Create a list of datasets from short notations.

This expands the `'ensemble'` and `'sub_experiment'` facets in the dataset definition if they are ranges.

For example `'ensemble'='r(1:3)i1p1f1'` will be expanded to three datasets, with `'ensemble'` values `'r1i1p1f1'`, `'r2i1p1f1'`, `'r3i1p1f1'`.

Returns
The datasets.

Return type
list[*Dataset*]

static from_recipe(*recipe: Path | str | dict*, *session:* Session) → list[*Dataset*]

Read datasets from a recipe.

Parameters

- **recipe** (*Path | str | dict*) – *Recipe* to load the datasets from. The value provided here should be either a path to a file, a recipe file that has been loaded using e.g. `yaml.safe_load()`, or an `str` that can be loaded using `yaml.safe_load()`.
- **session** (Session) – Datasets to use in the recipe.

Returns
A list of datasets.

Return type
list[*Dataset*]

load() → Cube

Load dataset.

Raises
InputFilesNotFound – When no files were found.

Returns
An `iris` cube with the data corresponding the the dataset.

Return type
iris.cube.Cube

property minimal_facets: Dict[str, str | Sequence[str] | Number]

Return a dictionary with the persistent facets.

property session: *Session*

A *esmvalcore.config.Session* associated with the dataset.

set_facet(*key: str*, *value: str | Sequence[str] | Number*, *persist: bool = True*)

Set facet.

Parameters

- **key** (*str*) – The name of the facet.
- **value** (*str | Sequence[str] | Number*) – The value of the facet.
- **persist** (*bool*) – When writing a dataset to a recipe, only persistent facets will get written.

set_version() → None

Set the `'version'` facet based on the available data.

Return type
None

summary(*shorten: bool = False*) → str

Summarize the content of dataset.

Parameters
shorten (*bool*) – Shorten the summary.

Returns
A summary describing the dataset.

Return type
str

esmvalcore.dataset.**INHERITED_FACETS: list[str] = ['dataset', 'domain', 'driver', 'grid', 'project', 'timerange']**

Inherited facets.

Supplementary datasets created based on the available files using the *Dataset.from_files()* method will inherit the values of these facets from the main dataset.

esmvalcore.dataset.**datasets_to_recipe**(*datasets: Iterable[Dataset]*, *recipe: Path | str | dict[str, Any] | None = None*) → dict

Create or update a recipe from datasets.

Parameters

- **datasets** (*Iterable[Dataset]*) – Datasets to use in the recipe.
- **recipe** (*Path | str | dict[str, Any] | None*) – *Recipe* to load the datasets from. The value provided here should be either a path to a file, a recipe file that has been loaded using e.g. `yaml.safe_load()`, or an `str` that can be loaded using `yaml.safe_load()`.

Return type
dict

Examples

See *Composing recipes* for example use cases.

Returns
The recipe with the datasets. To convert the `dict` to a *recipe*, use e.g. `yaml.safe_dump()`.

Return type
dict

Raises
RecipeError – Raised when a dataset is missing the `diagnostic` facet.

Parameters

- **datasets** (*Iterable[Dataset]*) –
- **recipe** (*Path | str | dict[str, Any] | None*) –

CHAPTER

THIRTYTWO

FIND AND DOWNLOAD FILES FROM ESGF

This module provides the function *esmvalcore.esgf.find_files()* for searching for files on ESGF using the ESMValTool vocabulary. It returns *esmvalcore.esgf.ESGFFile* objects, which have a convenient *esmvalcore.esgf.ESGFFile.download()* method for downloading the files.

See *ESGF configuration* for instructions on configuring this module.

32.1 esmvalcore.esgf

esmvalcore.esgf.**find_files**(*, *project*, *short_name*, *dataset*, ***facets*)

Search for files on ESGF.

Parameters

- **project** (*str*) – Choose from CMIP3, CMIP5, CMIP6, CORDEX, or obs4MIPs.
- **short_name** (*str*) – The name of the variable.
- **dataset** (*str*) – The name of the dataset.
- ****facets** (*Union[str, list[str]]*) – Any other search facets. An `'*'` can be used to match any value. By default, only the latest version of a file will be returned. To select all versions use `version='*'` while other omitted facets will default to `'*'`. It is also possible to specify multiple values for a facet, e.g. `exp=['historical', 'ssp585']` will match any file that belongs to either the historical or ssp585 experiment. The `timerange` facet can be specified in ISO 8601 format.

Note: A value of `timerange='*'` is supported, but combining a `'*'` with a time or period *as supported in the recipe* is currently not supported and will return all found files.

Examples

Examples of how to use this function for all supported projects.

Search for a CMIP3 dataset:

```
>>> find_files(
...     project='CMIP3',
...     frequency='mon',
...     short_name='tas',
...     dataset='cccma_cgcm3_1',
```

(continues on next page)

(continued from previous page)

```
...     exp='historical',
...     ensemble='run1',
... )
[ESGFFile:cmip3/CCCma/cccma_cgcm3_1/historical/mon/atmos/run1/tas/v1/tas_a1_20c3m_1_
↪cgcm3.1_t47_1850_2000.nc]
```

Search for a CMIP5 dataset:

```
>>> find_files(
...     project='CMIP5',
...     mip='Amon',
...     short_name='tas',
...     dataset='inmcm4',
...     exp='historical',
...     ensemble='r1i1p1',
... )
[ESGFFile:cmip5/output1/INM/inmcm4/historical/mon/atmos/Amon/r1i1p1/v20130207/tas_
↪Amon_inmcm4_historical_r1i1p1_185001-200512.nc]
```

Search for a CMIP6 dataset:

```
>>> find_files(
...     project='CMIP6',
...     mip='Amon',
...     short_name='tas',
...     dataset='CanESM5',
...     exp='historical',
...     ensemble='r1i1p1f1',
... )
[ESGFFile:CMIP6/CMIP/CCCma/CanESM5/historical/r1i1p1f1/Amon/tas/gn/v20190429/tas_
↪Amon_CanESM5_historical_r1i1p1f1_gn_185001-201412.nc]
```

Search for a CORDEX dataset and limit the search results to files containing data to the years in the range 1990-2000:

```
>>> find_files(
...     project='CORDEX',
...     frequency='mon',
...     dataset='COSMO-crCLIM-v1-1',
...     short_name='tas',
...     exp='historical',
...     ensemble='r1i1p1',
...     domain='EUR-11',
...     driver='MPI-M-MPI-ESM-LR',
...     timerange='1990/2000',
... )
[ESGFFile:cordex/output/EUR-11/CLMcom-ETH/MPI-M-MPI-ESM-LR/historical/r1i1p1/COSMO-
↪crCLIM-v1-1/v1/mon/tas/v20191219/tas_EUR-11_MPI-M-MPI-ESM-LR_historical_r1i1p1_
↪CLMcom-ETH-COSMO-crCLIM-v1-1_v1_mon_198101-199012.nc,
 ESGFFile:cordex/output/EUR-11/CLMcom-ETH/MPI-M-MPI-ESM-LR/historical/r1i1p1/COSMO-
↪crCLIM-v1-1/v1/mon/tas/v20191219/tas_EUR-11_MPI-M-MPI-ESM-LR_historical_r1i1p1_
↪CLMcom-ETH-COSMO-crCLIM-v1-1_v1_mon_199101-200012.nc]
```

Search for an obs4MIPs dataset:

```
>>> find_files(
...     project='obs4MIPs',
...     frequency='mon',
...     dataset='CERES-EBAF',
...     short_name='rsutcs',
... )
[ESGFFile:obs4MIPs/NASA-LaRC/CERES-EBAF/atmos/mon/v20160610/rsutcs_CERES-EBAF_L3B_
↪Ed2-8_200003-201404.nc]
```

Search for any ensemble member:

```
>>> find_files(
...     project='CMIP6',
...     mip='Amon',
...     short_name='tas',
...     dataset='BCC-CSM2-MR',
...     exp='historical',
...     ensemble='*',
... )
[ESGFFile:CMIP6/CMIP/BCC/BCC-CSM2-MR/historical/r1i1p1f1/Amon/tas/gn/v20181126/tas_
↪Amon_BCC-CSM2-MR_historical_r1i1p1f1_gn_185001-201412.nc,
 ESGFFile:CMIP6/CMIP/BCC/BCC-CSM2-MR/historical/r2i1p1f1/Amon/tas/gn/v20181115/tas_
↪Amon_BCC-CSM2-MR_historical_r2i1p1f1_gn_185001-201412.nc,
 ESGFFile:CMIP6/CMIP/BCC/BCC-CSM2-MR/historical/r3i1p1f1/Amon/tas/gn/v20181119/tas_
↪Amon_BCC-CSM2-MR_historical_r3i1p1f1_gn_185001-201412.nc]
```

Search for all available versions of a file:

```
>>> find_files(
...     project='CMIP5',
...     mip='Amon',
...     short_name='tas',
...     dataset='CCSM4',
...     exp='historical',
...     ensemble='r1i1p1',
...     version='*',
... )
[ESGFFile:cmip5/output1/NCAR/CCSM4/historical/mon/atmos/Amon/r1i1p1/v20121031/tas_
↪Amon_CCSM4_historical_r1i1p1_185001-200512.nc,
 ESGFFile:cmip5/output1/NCAR/CCSM4/historical/mon/atmos/Amon/r1i1p1/v20130425/tas_
↪Amon_CCSM4_historical_r1i1p1_185001-200512.nc,
 ESGFFile:cmip5/output1/NCAR/CCSM4/historical/mon/atmos/Amon/r1i1p1/v20160829/tas_
↪Amon_CCSM4_historical_r1i1p1_185001-200512.nc]
```

Search for a specific version of a file:

```
>>> find_files(
...     project='CMIP5',
...     mip='Amon',
...     short_name='tas',
...     dataset='CCSM4',
...     exp='historical',
```

(continues on next page)

(continued from previous page)

```
...     ensemble='r1i1p1',
...     version='v20130425',
... )
[ESGFFile:cmip5/output1/NCAR/CCSM4/historical/mon/atmos/Amon/r1i1p1/v20130425/tas_
↪Amon_CCSM4_historical_r1i1p1_185001-200512.nc]
```

Returns
A list of files that have been found.

Return type
list of *ESGFFile*

esmvalcore.esgf.**download**(*files*, *dest_folder*, *n_jobs=4*)

Download multiple ESGFFiles in parallel.

Parameters

- **files** (list of *ESGFFile*) – The files to download.
- **dest_folder** (*Path*) – The destination folder.
- **n_jobs** (*int*) – The number of files to download in parallel.

Raises
DownloadError: – Raised if one or more files failed to download.

class esmvalcore.esgf.**ESGFFile**(*results*)

Bases: object

File on the ESGF.

This is the object returned by *esmvalcore.esgf.find_files()*.

dataset

The name of the dataset that the file is part of.

Type
str

facets

Facets describing the file.

Type
dict[str,str]

name

The name of the file.

Type
str

size

The size of the file in bytes.

Type
int

urls

The URLs where the file can be downloaded.

Type
list[str]

Methods:

download(dest_folder)	Download the file.
local_file(dest_folder)	Return the path to the local file after download.

download(*dest_folder*)

Download the file.

Parameters
dest_folder (*Path*) – The destination folder.

Raises
DownloadError: – Raised if downloading the file failed.

Returns
The path where the file will be located after download.

Return type
LocalFile

local_file(*dest_folder*)

Return the path to the local file after download.

Parameters
dest_folder (*Path*) – The destination folder.

Returns
The path where the file will be located after download.

Return type
LocalFile

32.2 esmvalcore.esgf.facets

Module containing mappings from our names to ESGF names.

Data:

DATASET_MAP	Cache for the mapping between recipe/filesystem and ESGF dataset names.
FACETS	Mapping between the recipe and ESGF facet names.

Functions:

create_dataset_map()	Create the DATASET_MAP from recipe datasets to ESGF dataset names.

esmvalcore.esgf.facets.**DATASET_MAP** = {'CMIP3': {}, 'CMIP5': {'ACCESS1-0': 'ACCESS1.0', 'ACCESS1-3': 'ACCESS1.3', 'CESM1-BGC': 'CESM1(BGC)', 'CESM1-CAM5': 'CESM1(CAM5)', 'CESM1-CAM5-1-FV2': 'CESM1(CAM5.1,FV2)', 'CESM1-FASTCHEM': 'CESM1(FASTCHEM)', 'CESM1-WACCM': 'CESM1(WACCM)', 'CSIRO-Mk3-6-0': 'CSIRO-Mk3.6.0', 'GFDL-CM2p1': 'GFDL-CM2.1', 'MRI-AGCM3-2H': 'MRI-AGCM3.2H', 'MRI-AGCM3-2S': 'MRI-AGCM3.2S', 'bcc-csm1-1': 'BCC-CSM1.1', 'bcc-csm1-1-m': 'BCC-CSM1.1(m)', 'fio-esm': 'FIO-ESM', 'inmcm4': 'INM-CM4'}, 'CMIP6': {}, 'CORDEX': {}, 'obs4MIPs': {}}

Cache for the mapping between recipe/filesystem and ESGF dataset names.

esmvalcore.esgf.facets.**FACETS** = {'CMIP3': {'dataset': 'model', 'ensemble': 'ensemble', 'exp': 'experiment', 'frequency': 'time_frequency', 'short_name': 'variable'}, 'CMIP5': {'dataset': 'model', 'ensemble': 'ensemble', 'exp': 'experiment', 'frequency': 'time_frequency', 'institute': 'institute', 'mip': 'cmor_table', 'product': 'product', 'short_name': 'variable'}, 'CMIP6': {'activity': 'activity_drs', 'dataset': 'source_id', 'ensemble': 'member_id', 'exp': 'experiment_id', 'grid': 'grid_label', 'institute': 'institution_id', 'mip': 'table_id', 'short_name': 'variable'}, 'CORDEX': {'dataset': 'rcm_name', 'domain': 'domain', 'driver': 'driving_model', 'ensemble': 'ensemble', 'exp': 'experiment', 'frequency': 'time_frequency', 'institute': 'institute', 'product': 'product', 'short_name': 'variable'}, 'obs4MIPs': {'dataset': 'source_id', 'frequency': 'time_frequency', 'institute': 'institute', 'short_name': 'variable'}}

Mapping between the recipe and ESGF facet names.

esmvalcore.esgf.facets.**create_dataset_map**()

Create the DATASET_MAP from recipe datasets to ESGF dataset names.

Run *python -m esmvalcore.esgf.facets* to print an up to date map.

CHAPTER

THIRTYTHREE

EXCEPTIONS

Exceptions that may be raised by ESMValCore.

Exceptions:

ESMValCoreDeprecationWarning	Custom deprecation warning.
Error	Base class from which other exceptions are derived.
InputFilesNotFound(msg)	Files that are required to run the recipe have not been found.
InvalidConfigParameter	Config parameter is invalid.
MissingConfigParameter	Config parameter is missing.
RecipeError(msg)	Recipe contains an error.
SuppressedError	Errors subclassed from SuppressedError hide the full traceback.

exception esmvalcore.exceptions.**ESMValCoreDeprecationWarning**

Bases: UserWarning

Custom deprecation warning.

exception esmvalcore.exceptions.**Error**

Bases: Exception

Base class from which other exceptions are derived.

exception esmvalcore.exceptions.**InputFilesNotFound**(*msg*)

Bases: *RecipeError*

Files that are required to run the recipe have not been found.

exception esmvalcore.exceptions.**InvalidConfigParameter**

Bases: *Error*, *SuppressedError*

Config parameter is invalid.

exception esmvalcore.exceptions.**MissingConfigParameter**

Bases: UserWarning

Config parameter is missing.

exception esmvalcore.exceptions.**RecipeError**(*msg*)

Bases: *Error*

Recipe contains an error.

exception esmvalcore.exceptions.**SuppressedError**

Bases: Exception

Errors subclassed from SuppressedError hide the full traceback.

This can be used for simple user-facing errors that do not need the full traceback.

CHAPTER

THIRTYFOUR

IRIS HELPER FUNCTIONS

Auxiliary functions for `iris`.

Functions:

add_leading_dim_to_cube(cube, dim_coord)	Add new leading dimension to cube.
date2num(date, unit[, dtype])	Convert datetime object into numeric value with requested dtype.
merge_cube_attributes(cubes[, delimiter])	Merge attributes of all given cubes in-place.

esmvalcore.iris_helpers.**add_leading_dim_to_cube**(*cube*, *dim_coord*)

Add new leading dimension to cube.

An input cube with shape `(x, ..., z)` will be transformed to a cube with shape `(w, x, ..., z)` where `w` is the length of `dim_coord`. Note that the data is broadcasted to the new shape.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **dim_coord** (*iris.coords.DimCoord*) – Dimensional coordinate that is used to describe the new leading dimension. Needs to be 1D.

Returns

Transformed input cube with new leading dimension.

Return type

iris.cube.Cube

Raises

CoordinateMultiDimError – `dim_coord` is not 1D.

esmvalcore.iris_helpers.**date2num**(*date*, *unit*, *dtype=<class 'numpy.float64'>*)

Convert datetime object into numeric value with requested dtype.

This is a custom version of `cf_units.Unit.date2num()` that guarantees the correct dtype for the return value.

Parameters

- **date** (`datetime.datetime` or `cftime.datetime`) –
- **unit** (`cf_units.Unit`) –
- **dtype** (*a numpy dtype*) –

Returns

The return value of `unit.date2num` with the requested dtype.

Return type
numpy.ndarray of type *dtype*

esmvalcore.iris_helpers.**merge_cube_attributes**(*cubes: Sequence[Cube]*, *delimiter: str = ' '*) → None

Merge attributes of all given cubes in-place.

After this operation, the attributes of all given cubes are equal. This is useful for operations that combine cubes, such as iris.cube.CubeList.merge_cube() or iris.cube.CubeList.concatenate_cube().

Note: This function differs from iris.util.equalise_attributes() in this respect that it does not delete attributes that are not identical but rather concatenates them (sorted) using the given `delimiter`. E.g., the attributes `exp: historical` and `exp: ssp585` end up as `exp: historical ssp585` using the default `delimiter = ' '`.

Parameters

- **cubes** (*Sequence[Cube]*) – Input cubes whose attributes will be modified in-place.
- **delimiter** (*str*) – Delimiter that is used to concatenate non-identical attributes.

Return type
None

CHAPTER

THIRTYFIVE

FIND FILES ON THE LOCAL FILESYSTEM

Find files on the local filesystem.

Classes:

LocalFile(*args, **kwargs)	File on the local filesystem.

Functions:

find_files(*[, debug])	Find files on the local filesystem.

class esmvalcore.local.**LocalFile**(**args*, ***kwargs*)

Bases: PosixPath

File on the local filesystem.

Attributes:

facets	Facets describing the file.

property facets: Dict[str, str | Sequence[str] | Number]

Facets describing the file.

Note: When using *find_files()*, facets are read from the directory structure. Facets stored in filenames are not yet supported.

esmvalcore.local.**find_files**(*, *debug: bool = False*, ***facets: str | Sequence[str] | Number*) → list[*LocalFile*] | tuple[list[*LocalFile*], list[Path]]

Find files on the local filesystem.

The directories that are searched for files are defined in *esmvalcore.config.CFG* under the `'rootpath'` key using the directory structure defined under the `'drs'` key. If `esmvalcore.config.CFG['rootpath']` contains a key that matches the value of the `project` facet, those paths will be used. If there is no project specific key, the directories in `esmvalcore.config.CFG['rootpath']['default']` will be searched.

See *Input data* for extensive instructions on configuring ESMValCore so it can find files locally.

Parameters

- **debug** (*bool*) – When debug is set to `True`, the function will return a tuple with the first element containing the files that were found and the second element containing the `glob.glob()` patterns that were used to search for files.
- ****facets** (*str | Sequence[str] | Number*) – Facets used to search for files. An `'*'` can be used to match any value. By default, only the latest version of a file will be returned. To select all versions use `version='*'`. It is also possible to specify multiple values for a facet, e.g. `exp=['historical', 'ssp585']` will match any file that belongs to either the historical or ssp585 experiment. The `timerange` facet can be specified in ISO 8601 format.

Return type
list[LocalFile] | tuple[list[LocalFile], list[*Path*]]

Note: A value of `timerange='*'` is supported, but combining a `'*'` with a time or period *as supported in the recipe* is currently not supported and will return all found files.

Examples

Search for files containing surface air temperature from any CMIP6 model for the historical experiment:

```
>>> esmvalcore.local.find_files(
...     project='CMIP6',
...     activity='CMIP',
...     mip='Amon',
...     short_name='tas',
...     exp='historical',
...     dataset='*',
...     ensemble='*',
...     grid='*',
...     institute='*',
... )
[LocalFile('/home/bandela/climate_data/CMIP6/CMIP/BCC/BCC-ESM1/historical/r1i1p1f1/
↪Amon/tas/gn/v20181214/tas_Amon_BCC-ESM1_historical_r1i1p1f1_gn_185001-201412.nc')]
```

Returns
The files that were found.

Return type
list[*LocalFile*]

Parameters

- **debug** (*bool*) –
- **facets** (*str | Sequence[str] | Number*) –

CHAPTER

THIRTYSIX

PREPROCESSOR FUNCTIONS

esmvalcore.preprocessor.**DEFAULT_ORDER = ('fix_file', 'load', 'fix_metadata', 'concatenate', 'cmor_check_metadata', 'clip_timerange', 'fix_data', 'cmor_check_data', 'add_supplementary_variables', 'derive', 'extract_time', 'extract_season', 'extract_month', 'resample_hours', 'resample_time', 'extract_levels', 'weighting_landsea_fraction', 'mask_landsea', 'mask_glaciated', 'mask_landseaice', 'regrid', 'extract_coordinate_points', 'extract_point', 'extract_location', 'mask_multimodel', 'mask_fillvalues', 'mask_above_threshold', 'mask_below_threshold', 'mask_inside_range', 'mask_outside_range', 'clip', 'rolling_window_statistics', 'extract_region', 'extract_shape', 'extract_volume', 'extract_trajectory', 'extract_transect', 'detrend', 'extract_named_regions', 'axis_statistics', 'depth_integration', 'area_statistics', 'volume_statistics', 'amplitude', 'zonal_statistics', 'meridional_statistics', 'accumulate_coordinate', 'hourly_statistics', 'daily_statistics', 'monthly_statistics', 'seasonal_statistics', 'annual_statistics', 'decadal_statistics', 'climate_statistics', 'anomalies', 'regrid_time', 'timeseries_filter', 'linear_trend', 'linear_trend_stderr', 'convert_units', 'ensemble_statistics', 'multi_model_statistics', 'bias', 'remove_supplementary_variables', 'save')**

By default, preprocessor functions are applied in this order.

Preprocessor module.

Functions:

accumulate_coordinate(cube, coordinate)	Weight data using the bounds from a given coordinate.
add_supplementary_variables(cube, ...)	Add ancillary variables and/or cell measures.
amplitude(cube, coords)	Calculate amplitude of cycles by aggregating over coordinates.
annual_statistics(cube[, operator])	Compute annual statistics.
anomalies(cube, period[, reference, ...])	Compute anomalies using a mean with the specified granularity.
area_statistics(cube, operator, ...)	Apply a statistical operator in the horizontal plane.
axis_statistics(cube, axis, operator, ...)	Perform statistics along a given axis.
bias(products[, bias_type, ...])	Calculate biases.
climate_statistics(cube[, operator, period, ...])	Compute climate statistics with the specified granularity.
clip(cube[, minimum, maximum])	Clip values at a specified minimum and/or maximum value.
clip_timerange(cube, timerange)	Extract time range with a resolution up to seconds.
cmor_check_data(cube, cmor_table, mip, ...)	Check if data conforms to variable's CMOR definition.
cmor_check_metadata(cube, cmor_table, mip, ...)	Check if metadata conforms to variable's CMOR definition.

continues on next page

Table 1 – continued from previous page

concatenate(cubes[, check_level])	Concatenate all cubes after fixing metadata.
convert_units(cube, units)	Convert the units of a cube to new ones.
daily_statistics(cube[, operator])	Compute daily statistics.
decadal_statistics(cube[, operator])	Compute decadal statistics.
depth_integration(cube)	Determine the total sum over the vertical component.
derive(cubes, short_name, long_name, units)	Derive variable.
detrend(cube[, dimension, method])	Detrend data along a given dimension.
ensemble_statistics(products, statistics, ...)	Entry point for ensemble statistics.
extract_coordinate_points(cube, definition, ...)	Extract points from any coordinate with interpolation.
extract_levels(cube, levels, scheme[, ...])	Perform vertical interpolation.
extract_location(cube, location, scheme)	Extract a point using a location name, with interpolation.
extract_month(cube, month)	Slice cube to get only the data belonging to a specific month.
extract_named_regions(cube, regions)	Extract a specific named region.
extract_point(cube, latitude, longitude, scheme)	Extract a point, with interpolation.
extract_region(cube, start_longitude, ...)	Extract a region from a cube.
extract_season(cube, season)	Slice cube to get only the data belonging to a specific season.
extract_shape(cube, shapefile[, method, ...])	Extract a region defined by a shapefile using masking.
extract_time(cube, start_year, start_month, ...)	Extract a time range from a cube.
extract_trajectory(cube, latitudes, longitudes)	Extract data along a trajectory.
extract_transect(cube[, latitude, longitude])	Extract data along a line of constant latitude or longitude.
extract_volume(cube, z_min, z_max[, ...])	Subset a cube based on a range of values in the z-coordinate.
fix_data(cube, short_name, project, dataset, mip)	Fix cube data if fixes are required.
fix_file(file, short_name, project, dataset, ...)	Fix files before ESMValTool can load them.
fix_metadata(cubes, short_name, project, ...)	Fix cube metadata if fixes are required.
hourly_statistics(cube, hours[, operator])	Compute hourly statistics.
linear_trend(cube[, coordinate])	Calculate linear trend of data along a given coordinate.
linear_trend_stderr(cube[, coordinate])	Calculate standard error of linear trend along a given coordinate.
load(file[, ignore_warnings])	Load iris cubes from string or Path objects.
mask_above_threshold(cube, threshold)	Mask above a specific threshold value.
mask_below_threshold(cube, threshold)	Mask below a specific threshold value.
mask_fillvalues(products, threshold_fraction)	Compute and apply a multi-dataset fillvalues mask.
mask_glaciated(cube[, mask_out])	Mask out glaciated areas.
mask_inside_range(cube, minimum, maximum)	Mask inside a specific threshold range.
mask_landsea(cube, mask_out)	Mask out either land mass or sea (oceans, seas and lakes).
mask_landseaice(cube, mask_out)	Mask out either landsea (combined) or ice.
mask_multimodel(products)	Apply common mask to all datasets (using logical OR).
mask_outside_range(cube, minimum, maximum)	Mask outside a specific threshold range.
meridional_statistics(cube, operator, ...)	Compute meridional statistics.
monthly_statistics(cube[, operator])	Compute monthly statistics.
multi_model_statistics(products, span, ...)	Compute multi-model statistics.
regrid(cube, target_grid, scheme[, ...])	Perform horizontal regridding.
regrid_time(cube, frequency)	Align time axis for cubes so they can be subtracted.
remove_supplementary_variables(cube)	Remove supplementary variables.
resample_hours(cube, interval[, offset])	Convert x-hourly data to y-hourly by eliminating extra timesteps.
resample_time(cube[, month, day, hour])	Change frequency of data by resampling it.
rolling_window_statistics(cube, coordinate, ...)	Compute rolling-window statistics over a coordinate.

continues on next page

Table 1 – continued from previous page

save(cubes, filename[, optimize_access, ...])	Save iris cubes to file.
seasonal_statistics(cube[, operator, seasons])	Compute seasonal statistics.
timeseries_filter(cube, window, span[, ...])	Apply a timeseries filter.
volume_statistics(cube, operator, ...)	Apply a statistical operation over a volume.
weighting_landsea_fraction(cube, area_type)	Weight fields using land or sea fraction.
zonal_statistics(cube, operator, ...)	Compute zonal statistics.

esmvalcore.preprocessor.**accumulate_coordinate**(*cube*, *coordinate*)

Weight data using the bounds from a given coordinate.

The resulting cube will then have units given by `cube_units * coordinate_units`.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube for the flux
- **coordinate** (*str*) – Name of the coordinate that will be used as weights.

Returns
Cube with the aggregated data

Return type
iris.cube.Cube

Raises

- **ValueError** – If the coordinate is not found in the cube.
- **NotImplementedError** – If the coordinate is multidimensional.

esmvalcore.preprocessor.**add_supplementary_variables**(*cube: Cube*, *supplementary_cubes: Iterable[Cube]*) → Cube

Add ancillary variables and/or cell measures.

Parameters

- **cube** (*Cube*) – Cube to add to.
- **supplementary_cubes** (*Iterable[Cube]*) – Iterable of cubes containing the supplementary variables.

Returns
Cube with added ancillary variables and/or cell measures.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**amplitude**(*cube*, *coords*)

Calculate amplitude of cycles by aggregating over coordinates.

Note: The amplitude is calculated as *peak-to-peak* amplitude (difference between maximum and minimum value of the signal). Other amplitude types are currently not supported.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.

- **coords** (*str or list of str*) – Coordinates over which is aggregated. For example, use `'year'` to extract the annual cycle amplitude for each year in the data or `['day_of_year', 'year']` to extract the diurnal cycle amplitude for each individual day in the data. If the coordinates are not found in `cube`, try to add it via `iris.coord_categorisation` (at the moment, this only works for the temporal coordinates `day_of_month`, `day_of_year`, `hour`, `month`, `month_fullname`, `month_number`, `season`, `season_number`, `season_year`, `weekday`, `weekday_fullname`, `weekday_number` or `year`.

Returns
Amplitudes.

Return type
iris.cube.Cube

Raises
iris.exceptions.CoordinateNotFoundError – A coordinate is not found in `cube` and cannot be added via `iris.coord_categorisation`.

esmvalcore.preprocessor.**annual_statistics**(*cube: Cube*, *operator: str = 'mean'*, ***operator_kwargs*) → Cube

Compute annual statistics.

Note that this function does not weight the annual mean if uneven time periods are present. Ie, all data inside the year are treated equally.

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Annual statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**anomalies**(*cube: Cube*, *period: str*, *reference: dict | None = None*, *standardize: bool = False*, *seasons: Iterable[str] = ('DJF', 'MAM', 'JJA', 'SON')*) → Cube

Compute anomalies using a mean with the specified granularity.

Computes anomalies based on hourly, daily, monthly, seasonal or yearly means for the full available period.

Parameters

- **cube** (*Cube*) – Input cube.
- **period** (*str*) – Period to compute the statistic over. Available periods: *full*, *season*, *seasonal*, *monthly*, *month*, *mon*, *daily*, *day*, *hourly*, *hour*, *hr*.
- **reference** (*optional*) – Period of time to use a reference, as needed for the *extract_time()* preprocessor function. If `None`, all available data is used as a reference.
- **standardize** (*optional*) – If `True` standardized anomalies are calculated.
- **seasons** (*optional*) – Seasons to use if needed. Defaults to ('DJF', 'MAM', 'JJA', 'SON').

Returns
Anomalies cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**area_statistics**(*cube: Cube*, *operator: str*, ***operator_kwargs*) → Cube

Apply a statistical operator in the horizontal plane.

We assume that the horizontal directions are ['longitude', 'latitude'].

This table shows a list of supported operators. All operators that support weights are by default weighted with the grid cell areas. Note that for area-weighted sums, the units of the resulting cube will be multiplied by m^2.

Parameters

- **cube** (*Cube*) – Input cube. The input cube should have a `iris.coords.CellMeasure` named `'cell_area'`, unless it has regular 1D latitude and longitude coordinates so the cell areas can be computed using `iris.analysis.cartography.area_weights()`.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Collapsed cube.

Return type
iris.cube.Cube

Raises
iris.exceptions.CoordinateMultiDimError – Cube has irregular or unstructured grid but supplementary variable *cell_area* is not available.

esmvalcore.preprocessor.**axis_statistics**(*cube: Cube*, *axis: str*, *operator: str*, ***operator_kwargs*) → Cube

Perform statistics along a given axis.

Operates over an axis direction.

Note: The *mean*, *sum* and *rms* operations are weighted by the corresponding coordinate bounds by default. For *sum*, the units of the resulting cube will be multiplied by corresponding coordinate units.

Parameters

- **cube** (*Cube*) – Input cube.
- **axis** (*str*) – Direction over where to apply the operator. Possible values are *x*, *y*, *z*, *t*.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Collapsed cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**bias**(*products*, *bias_type='absolute'*, *denominator_mask_threshold=0.001*, *keep_reference_dataset=False*)

Calculate biases.

Notes

This preprocessor requires a reference dataset. For this, exactly one input dataset needs to have the facet `reference_for_bias: true` defined in the recipe. In addition, all input datasets need to have identical dimensional coordinates. This can for example be ensured with the preprocessors *esmvalcore.preprocessor.regrid()* and/or *esmvalcore.preprocessor.regrid_time()*.

Parameters

- **products** (*set of esmvalcore.preprocessor.PreprocessorFile*) – Input datasets. Exactly one datasets needs the facet `reference_for_bias: true`.
- **bias_type** (*str, optional (default: 'absolute')*) – Bias type that is calculated. Must be one of `'absolute'` (dataset - ref) or `'relative'` ((dataset - ref) / ref).
- **denominator_mask_threshold** (*float, optional (default: 1e-3)*) – Threshold to mask values close to zero in the denominator (i.e., the reference dataset) during the calculation of relative biases. All values in the reference dataset with absolute value less than the given threshold are masked out. This setting is ignored when `bias_type` is set to `'absolute'`. Please note that for some variables with very small absolute values (e.g., carbon cycle fluxes, which are usually $< 10^{-6}$ kg m $^{-2}$ s $^{-1}$) it is absolutely essential to change the default value in order to get reasonable results.
- **keep_reference_dataset** (*bool, optional (default: False)*) – If `True`, keep the reference dataset in the output. If `False`, drop the reference dataset.

Returns
Output datasets.

Return type
set of esmvalcore.preprocessor.PreprocessorFile

Raises
ValueError – Not exactly one input datasets contains the facet `reference_for_bias: true`; `bias_type` is not one of `'absolute'` or `'relative'`.

esmvalcore.preprocessor.**climate_statistics**(*cube: Cube*, *operator: str = 'mean'*, *period: str = 'full'*, *seasons: Iterable[str] = ('DJF', 'MAM', 'JJA', 'SON')*, ***operator_kwargs*) → Cube

Compute climate statistics with the specified granularity.

Computes statistics for the whole dataset. It is possible to get them for the full period or with the data grouped by hour, day, month or season.

Note: The *mean*, *sum* and *rms* operations over the *full* period are weighted by the time coordinate, i.e., the length of the time intervals. For *sum*, the units of the resulting cube will be multiplied by corresponding time units (e.g., days).

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- **period** (*str*) – Period to compute the statistic over. Available periods: *full*, *season*, *seasonal*, *monthly*, *month*, *mon*, *daily*, *day*, *hourly*, *hour*, *hr*.
- **seasons** (*Iterable[str]*) – Seasons to use if needed. Defaults to ('DJF', 'MAM', 'JJA', 'SON').
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Climate statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**clip**(*cube*, *minimum=None*, *maximum=None*)

Clip values at a specified minimum and/or maximum value.

Values lower than minimum are set to minimum and values higher than maximum are set to maximum.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be clipped
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns
clipped cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**clip_timerange**(*cube: Cube*, *timerange: str*) → Cube

Extract time range with a resolution up to seconds.

Parameters

- **cube** (*Cube*) – Input cube.
- **timerange** (*str*) – Time range in ISO 8601 format.

Returns
Sliced cube.

Return type
iris.cube.Cube

Raises
ValueError – Time ranges are outside the cube's time limits.

esmvalcore.preprocessor.**cmor_check_data**(*cube: Cube*, *cmor_table: str*, *mip: str*, *short_name: str*, *frequency: str | None = None*, *check_level:* CheckLevels = *CheckLevels.DEFAULT*) → Cube

Check if data conforms to variable's CMOR definition.

Parameters

- **cube** (*Cube*) – Data cube to check.

- **cmor_table** (*str*) – CMOR definitions to use (i.e., the variable's project).
- **mip** (*str*) – Variable's MIP.
- **short_name** (*str*) – Variable's short name
- **frequency** (*str | None*) – Data frequency. If not given, use the one from the CMOR table of the variable.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Returns
Checked cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**cmor_check_metadata**(*cube: Cube*, *cmor_table: str*, *mip: str*, *short_name: str*, *frequency: str | None = None*, *check_level: CheckLevels = CheckLevels.DEFAULT*) → Cube

Check if metadata conforms to variable's CMOR definition.

None of the checks at this step will force the cube to load the data.

Parameters

- **cube** (*Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use (i.e., the variable's project).
- **mip** (*str*) – Variable's MIP.
- **short_name** (*str*) – Variable's short name.
- **frequency** (*str | None*) – Data frequency. If not given, use the one from the CMOR table of the variable.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Returns
Checked cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**concatenate**(*cubes*, *check_level=CheckLevels.DEFAULT*)

Concatenate all cubes after fixing metadata.

Parameters

- **cubes** (*iterable of iris.cube.Cube*) – Data cubes to be concatenated
- **check_level** (CheckLevels) – Level of strictness of the checks in the concatenation.

Returns
cube – Resulting concatenated cube.

Return type
iris.cube.Cube

Raises
ValueError – Concatenation was not possible.

esmvalcore.preprocessor.**convert_units**(*cube*, *units*)

Convert the units of a cube to new ones.

This converts units of a cube.

Note: Allows special unit conversions which transforms one quantity to another (physically related) quantity. These quantities are identified via their `standard_name` and their `units` (units convertible to the ones defined are also supported). For example, this enables conversions between precipitation fluxes measured in `kg m-2 s-1` and precipitation rates measured in `mm day-1` (and vice versa).

Currently, the following special conversions are supported:

- `precipitation_flux` (`kg m-2 s-1`) – `lwe_precipitation_rate` (`mm day-1`)

Names in the list correspond to `standard_names` of the input data. Conversions are allowed from each quantity to any other quantity given in a bullet point. The corresponding target quantity is inferred from the desired target units. In addition, any other units convertible to the ones given are also supported (e.g., instead of `mm day-1`, `m s-1` is also supported).

Note that for precipitation variables, a water density of `1000 kg m-3` is assumed.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **units** (*str*) – New units in udunits form.

Returns
converted cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**daily_statistics**(*cube: Cube*, *operator: str = 'mean'*, ***operator_kwargs*) → Cube

Compute daily statistics.

Chunks time in daily periods and computes statistics over them;

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Daily statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**decadal_statistics**(*cube: Cube*, *operator: str = 'mean'*, ***operator_kwargs*) → Cube

Compute decadal statistics.

Note that this function does not weight the decadal mean if uneven time periods are present. Ie, all data inside the decade are treated equally.

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Decadal statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**depth_integration**(*cube: Cube*) → Cube

Determine the total sum over the vertical component.

Requires a 3D cube. The z-coordinate integration is calculated by taking the sum in the z direction of the cell contents multiplied by the cell thickness. The units of the resulting cube are multiplied by the z-coordinate units.

Parameters
cube (*Cube*) – Input cube.

Returns
Collapsed cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**derive**(*cubes*, *short_name*, *long_name*, *units*, *standard_name=None*)

Derive variable.

Parameters

- **cubes** (*iris.cube.CubeList*) – Includes all the needed variables for derivation defined in `get_required()`.
- **short_name** (*str*) – short_name
- **long_name** (*str*) – long_name
- **units** (*str*) – units
- **standard_name** (*str, optional*) – standard_name

Returns
The new derived variable.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**detrend**(*cube*, *dimension='time'*, *method='linear'*)

Detrend data along a given dimension.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **dimension** (*str*) – Dimension to detrend
- **method** (*str*) – Method to detrend. Available: linear, constant. See documentation of 'scipy.signal.detrend' for details

Returns
Detrended cube

Return type
iris.cube.Cube

esmvalcore.preprocessor.**ensemble_statistics**(*products: set[PreprocessorFile] | Iterable[Cube]*, *statistics: list[str]*, *output_products*, *span: str = 'overlap'*, *ignore_scalar_coords: bool = False*) → dict | set

Entry point for ensemble statistics.

An ensemble grouping is performed on the input products. The statistics are then computed calling the *esmvalcore.preprocessor.multi_model_statistics()* module, taking the grouped products as an input.

Parameters

- **products** (*set[PreprocessorFile] | Iterable[Cube]*) – Cubes (or products) over which the statistics will be computed.
- **statistics** (*list[str]*) – Statistical metrics to be computed, e.g. [mean, max]. Choose from the operators listed in the iris.analysis package. Percentiles can be specified like pXX.YY.
- **output_products** (*dict*) – For internal use only. A dict with statistics names as keys and preprocessorfiles as values. If products are passed as input, the statistics cubes will be assigned to these output products.
- **span** (*str*) – Overlap or full; if overlap, statitstics are computed on common time- span; if full, statistics are computed on full time spans, ignoring missing data.
- **ignore_scalar_coords** (*bool*) – If True, remove any scalar coordinate in the input datasets before merging the input cubes into the multi-dataset cube. The resulting multi-dataset cube will have no scalar coordinates (the actual input datasets will remain unchanged). If False, scalar coordinates will remain in the input datasets, which might lead to merge conflicts in case the input datasets have different scalar coordinates.

Returns
A dict of cubes or set of *output_products* depending on the type of *products*.

Return type
dict | set

See also:

esmvalcore.preprocessor.multi_model_statistics(), the

esmvalcore.preprocessor.**extract_coordinate_points**(*cube*, *definition*, *scheme*)

Extract points from any coordinate with interpolation.

Multiple points can also be extracted, by supplying an array of coordinates. The resulting point cube will match the respective coordinates to those of the input coordinates. If the input coordinate is a scalar, the dimension will be a scalar in the output cube.

Parameters

- **cube** (*cube*) – The source cube to extract a point from.
- **definition** (*dict(str, float or array of float)*) – The coordinate - values pairs to extract
- **scheme** (*str*) – The interpolation scheme. 'linear' or 'nearest'. No default.

Returns
Returns a cube with the extracted point(s), and with adjusted latitude and longitude coordinates (see above). If desired point outside values for at least one coordinate, this cube will have fully masked data.

Return type
Cube

Raises
ValueError: – If the interpolation scheme is not provided or is not recognised.

esmvalcore.preprocessor.**extract_levels**(*cube*, *levels*, *scheme*, *coordinate=None*, *rtol=1e-07*, *atol=None*)

Perform vertical interpolation.

Parameters

- **cube** (*iris.cube.Cube*) – The source cube to be vertically interpolated.
- **levels** (*ArrayLike*) – One or more target levels for the vertical interpolation. Assumed to be in the same S.I. units of the source cube vertical dimension coordinate. If the requested levels are sufficiently close to the levels of the cube, cube slicing will take place instead of interpolation.
- **scheme** (*str*) – The vertical interpolation scheme to use. Choose from 'linear', 'nearest', 'linear_extrapolate', 'nearest_extrapolate'.
- **coordinate** (*optional str*) – The coordinate to interpolate. If specified, pressure levels (if present) can be converted to height levels and vice versa using the US standard atmosphere. E.g. 'coordinate = altitude' will convert existing pressure levels (air_pressure) to height levels (altitude); 'coordinate = air_pressure' will convert existing height levels (altitude) to pressure levels (air_pressure).
- **rtol** (*float*) – Relative tolerance for comparing the levels in *cube* to the requested levels. If the levels are sufficiently close, the requested levels will be assigned to the cube and no interpolation will take place.
- **atol** (*float*) – Absolute tolerance for comparing the levels in *cube* to the requested levels. If the levels are sufficiently close, the requested levels will be assigned to the cube and no interpolation will take place. By default, *atol* will be set to 10^-7 times the mean value of the levels on the cube.

Returns
A cube with the requested vertical levels.

Return type
iris.cube.Cube

See also:

regrid
Perform horizontal regridding.

esmvalcore.preprocessor.**extract_location**(*cube*, *location*, *scheme*)

Extract a point using a location name, with interpolation.

Extracts a single location point from a cube, according to the interpolation scheme `scheme`.

The function just retrieves the coordinates of the location and then calls the `extract_point` preprocessor.

It can be used to locate cities and villages, but also mountains or other geographical locations.

Note: The geolocator needs a working internet connection.

Parameters

- **cube** (*cube*) – The source cube to extract a point from.
- **location** (*str*) – The reference location. Examples: 'mount everest', 'romania','new york, usa'
- **scheme** (*str*) – The interpolation scheme. 'linear' or 'nearest'. No default.

Returns

- *Returns a cube with the extracted point, and with adjusted*
- *latitude and longitude coordinates.*

Raises

- **ValueError:** – If location is not supplied as a preprocessor parameter.
- **ValueError:** – If scheme is not supplied as a preprocessor parameter.
- **ValueError:** – If given location cannot be found by the geolocator.

esmvalcore.preprocessor.**extract_month**(*cube: Cube*, *month: int*) → Cube

Slice cube to get only the data belonging to a specific month.

Parameters

- **cube** (*Cube*) – Original data
- **month** (*int*) – Month to extract as a number from 1 to 12.

Returns
Cube for specified month.

Return type
iris.cube.Cube

Raises
ValueError – Requested month is not present in the cube.

esmvalcore.preprocessor.**extract_named_regions**(*cube: Cube*, *regions: str | Iterable[str]*) → Cube

Extract a specific named region.

The region coordinate exist in certain CMIP datasets. This preprocessor allows a specific named regions to be extracted.

Parameters

- **cube** (*Cube*) – Input cube.
- **regions** (*str | Iterable[str]*) – A region or list of regions to extract.

Returns
Smaller cube.

Return type
iris.cube.Cube

Raises

- **ValueError** – regions is not list or tuple or set.

- **ValueError** – region not included in cube.

esmvalcore.preprocessor.**extract_point**(*cube*, *latitude*, *longitude*, *scheme*)

Extract a point, with interpolation.

Extracts a single latitude/longitude point from a cube, according to the interpolation scheme *scheme*.

Multiple points can also be extracted, by supplying an array of latitude and/or longitude coordinates. The resulting point cube will match the respective latitude and longitude coordinate to those of the input coordinates. If the input coordinate is a scalar, the dimension will be missing in the output cube (that is, it will be a scalar).

If the point to be extracted has at least one of the coordinate point values outside the interval of the cube's same coordinate values, then no extrapolation will be performed, and the resulting extracted cube will have fully masked data.

Parameters

- **cube** (*cube*) – The source cube to extract a point from.
- **latitude** (*float, or array of float*) – The latitude and longitude of the point.
- **longitude** (*float, or array of float*) – The latitude and longitude of the point.
- **scheme** (*str*) – The interpolation scheme. 'linear' or 'nearest'. No default.

Returns

Returns a cube with the extracted point(s), and with adjusted latitude and longitude coordinates (see above). If desired point outside values for at least one coordinate, this cube will have fully masked data.

Return type

Cube

Raises

ValueError: – If the interpolation scheme is None or unrecognized.

Examples

With a cube that has the coordinates

- latitude: [1, 2, 3, 4]
- longitude: [1, 2, 3, 4]
- **data values: [[[1, 2, 3, 4], [5, 6, ...], [...], [...],**
  ...]]]

```
>>> point = extract_point(cube, 2.5, 2.5, 'linear')
>>> point.data
array([ 8.5, 24.5, 40.5, 56.5])
```

Extraction of multiple points at once, with a nearest matching scheme. The values for 0.1 will result in masked values, since this lies outside the cube grid.

```
>>> point = extract_point(cube, [1.4, 2.1], [0.1, 1.1],
...                       'nearest')
>>> point.data.shape
(4, 2, 2)
>>> # x, y, z indices of masked values
>>> np.where(~point.data.mask)
```

(continues on next page)

(continued from previous page)

```
(array([0, 0, 1, 1, 2, 2, 3, 3]), array([0, 1, 0, 1, 0, 1, 0, 1]),
array([1, 1, 1, 1, 1, 1, 1, 1]))
>>> point.data[~point.data.mask].data
array([ 1,  5, 17, 21, 33, 37, 49, 53])
```

esmvalcore.preprocessor.**extract_region**(*cube: Cube*, *start_longitude: float*, *end_longitude: float*, *start_latitude: float*, *end_latitude: float*) → Cube

Extract a region from a cube.

Function that subsets a cube on a box (start_longitude, end_longitude, start_latitude, end_latitude).

Parameters

- **cube** (*Cube*) – Input data cube.
- **start_longitude** (*float*) – Western boundary longitude.
- **end_longitude** (*float*) – Eastern boundary longitude.
- **start_latitude** (*float*) – Southern Boundary latitude.
- **end_latitude** (*float*) – Northern Boundary Latitude.

Returns
Smaller cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**extract_season**(*cube: Cube*, *season: str*) → Cube

Slice cube to get only the data belonging to a specific season.

Parameters

- **cube** (*Cube*) – Original data
- **season** (*str*) – Season to extract. Available: DJF, MAM, JJA, SON and all sequentially correct combinations: e.g. JJAS

Returns
data cube for specified season.

Return type
iris.cube.Cube

Raises
ValueError – Requested season is not present in the cube.

esmvalcore.preprocessor.**extract_shape**(*cube: Cube*, *shapefile: str | Path*, *method: str = 'contains'*, *crop: bool = True*, *decomposed: bool = False*, *ids: list | dict | None = None*) → Cube

Extract a region defined by a shapefile using masking.

Note that this function does not work for shapes crossing the prime meridian or poles.

Parameters

- **cube** (*Cube*) – Input cube.
- **shapefile** (*str | Path*) – A shapefile defining the region(s) to extract. Also accepts the following strings to load special shapefiles:

- `'ar6'`: IPCC WG1 reference regions (v4) used in Assessment Report 6 (https://doi.org/10.5281/zenodo.5176260). Should be used in combination with a `dict` for the argument *ids*, e.g., `ids={'Acronym': ['GIC', 'WNA']}`.

- **method** (*str*) – Select all points contained by the shape or select a single representative point. Choose either *'contains'* or *'representative'*. If *'contains'* is used, but not a single grid point is contained by the shape, a representative point will be selected.
- **crop** (*bool*) – In addition to masking, crop the resulting cube using *extract_region()*. Data on irregular grids will not be cropped.
- **decomposed** (*bool*) – If set to *True*, the output cube will have an additional dimension *shape_id* describing the requested regions.
- **ids** (*list | dict | None*) – Shapes to be read from the shapefile. Can be given as:
  - `list`: IDs are assigned from the attributes *name*, *NAME*, *Name*, *id*, or *ID* (in that priority order; the first one available is used). If none of these attributes are available in the shapefile, assume that the given *ids* correspond to the reading order of the individual shapes, e.g., `ids=[0, 2]` corresponds to the first and third shape read from the shapefile. Note: An empty list is interpreted as *ids=None*.
  - `dict`: IDs (dictionary value; `list` of `str`) are assigned from attribute given as dictionary key (`str`). Only dictionaries with length 1 are supported. Example: `ids={'Acronym': ['GIC', 'WNA']}` for `shapefile='ar6'`.
  - *None*: select all available shapes from the shapefile.

Returns
Cube containing the extracted region.

Return type
iris.cube.Cube

See also:

extract_region
Extract a region from a cube.

esmvalcore.preprocessor.**extract_time**(*cube: Cube*, *start_year: int*, *start_month: int*, *start_day: int*, *end_year: int*, *end_month: int*, *end_day: int*) → Cube

Extract a time range from a cube.

Given a time range passed in as a series of years, months and days, it returns a time-extracted cube with data only within the specified time range.

Parameters

- **cube** (*Cube*) – Input cube.
- **start_year** (*int*) – Start year.
- **start_month** (*int*) – Start month.
- **start_day** (*int*) – Start day.
- **end_year** (*int*) – End year.
- **end_month** (*int*) – End month.
- **end_day** (*int*) – End day.

Returns
Sliced cube.

Return type
iris.cube.Cube

Raises
ValueError – Time ranges are outside the cube time limits.

esmvalcore.preprocessor.**extract_trajectory**(*cube: Cube*, *latitudes: Sequence[float]*, *longitudes: Sequence[float]*, *number_points: int = 2*) → Cube

Extract data along a trajectory.

latitudes and longitudes are the pairs of coordinates for two points. number_points is the number of points between the two points.

This version uses the expensive interpolate method, but it may be necceasiry for irregular grids.

If only two latitude and longitude coordinates are given, extract_trajectory will produce a cube will extrapolate along a line between those two points, and will add *number_points* points between the two corners.

If more than two points are provided, then extract_trajectory will produce a cube which has extrapolated the data of the cube to those points, and *number_points* is not needed.

Parameters

- **cube** (*Cube*) – Input cube.
- **latitudes** (*Sequence[float]*) – Latitude coordinates.
- **longitudes** (*Sequence[float]*) – Longitude coordinates.
- **number_points** (*optional*) – Number of points to extrapolate.

Returns
Collapsed cube.

Return type
iris.cube.Cube

Raises
ValueError – Latitude and longitude have different dimensions.

esmvalcore.preprocessor.**extract_transect**(*cube: Cube*, *latitude: None | float | Iterable[float] = None*, *longitude: None | float | Iterable[float] = None*) → Cube

Extract data along a line of constant latitude or longitude.

Both arguments, latitude and longitude, are treated identically. Either argument can be a single float, or a pair of floats, or can be left empty. The single float indicates the latitude or longitude along which the transect should be extracted. A pair of floats indicate the range that the transect should be extracted along the secondairy axis.

For instance *'extract_transect(cube, longitude=-28)'* will produce a transect along 28 West.

Also, *'extract_transect(cube, longitude=-28, latitude=[-50, 50])'* will produce a transect along 28 West between 50 south and 50 North.

This function is not yet implemented for irregular arrays - instead try the extract_trajectory function, but note that it is currently very slow. Alternatively, use the regrid preprocessor to regrid along a regular grid and then extract the transect.

Parameters

- **cube** (*Cube*) – Input cube.
- **latitude** (*optional*) – Transect latitude or range.
- **longitude** (*optional*) – Transect longitude or range.

Returns
Collapsed cube.

Return type
iris.cube.Cube

Raises

- **ValueError** – Slice extraction not implemented for irregular grids.
- **ValueError** – Latitude and longitude are both floats or lists; not allowed to slice on both axes at the same time.

esmvalcore.preprocessor.**extract_volume**(*cube: Cube*, *z_min: float*, *z_max: float*, *interval_bounds: str = 'open'*, *nearest_value: bool = False*) → Cube

Subset a cube based on a range of values in the z-coordinate.

Function that subsets a cube on a box of (z_min, z_max), (z_min, z_max], [z_min, z_max) or [z_min, z_max] Note that this requires the requested z-coordinate range to be the same sign as the iris cube. ie, if the cube has z-coordinate as negative, then z_min and z_max need to be negative numbers. If nearest_value is set to *False*, the extraction will be performed with the given z_min and z_max values. If nearest_value is set to *True*, the cube extraction will be performed taking into account the z_coord values that are closest to the z_min and z_max values.

Parameters

- **cube** (*Cube*) – Input cube.
- **z_min** (*float*) – Minimum depth to extract.
- **z_max** (*float*) – Maximum depth to extract.
- **interval_bounds** (*str*) – Sets left bound of the interval to either 'open', 'closed', 'left_closed' or 'right_closed'.
- **nearest_value** (*bool*) – Extracts considering the nearest value of z-coord to z_min and z_max.

Returns
z-coord extracted cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**fix_data**(*cube: Cube*, *short_name: str*, *project: str*, *dataset: str*, *mip: str*, *frequency: str | None = None*, *check_level: CheckLevels = CheckLevels.DEFAULT*, *session: Session | None = None*, ***extra_facets*) → Cube

Fix cube data if fixes are required.

This method assumes that metadata is already fixed and checked.

This method collects all the relevant fixes (including generic ones) for a given variable and applies them.

Parameters

- **cube** (*Cube*) – Cube to fix.
- **short_name** (*str*) – Variable's short name.
- **project** (*str*) – Project of the dataset.
- **dataset** (*str*) – Name of the dataset.
- **mip** (*str*) – Variable's MIP.

- **frequency** (*Optional[str]*) – Variable's data frequency, if available.
- **check_level** (CheckLevels) – Level of strictness of the checks.

  Deprecated since version 2.10.0: This option has been deprecated in ESMValCore version 2.10.0 and is scheduled for removal in version 2.12.0. Please use the functions *cmor_check_metadata()*, *cmor_check_data()*, or *cmor_check()* instead. This function will no longer perform CMOR checks. Fixes and CMOR checks have been clearly separated in ESMValCore version 2.10.0.
- **session** (*Optional[Session]*) – Current session which includes configuration and directory information.
- ****extra_facets** – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns
Fixed cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**fix_file**(*file: Path*, *short_name: str*, *project: str*, *dataset: str*, *mip: str*, *output_dir: Path*, *add_unique_suffix: bool = False*, *session: Session | None = None*, *frequency: str | None = None*, ***extra_facets*) → str | Path

Fix files before ESMValTool can load them.

These fixes are only for issues that prevent iris from loading the cube or that cannot be fixed after the cube is loaded.

Original files are not overwritten.

Parameters

- **file** (*Path*) – Path to the original file.
- **short_name** (*str*) – Variable's short name.
- **project** (*str*) – Project of the dataset.
- **dataset** (*str*) – Name of the dataset.
- **mip** (*str*) – Variable's MIP.
- **output_dir** (*Path*) – Output directory for fixed files.
- **add_unique_suffix** (*bool*) – Adds a unique suffix to *output_dir* for thread safety.
- **session** (*Optional[Session]*) – Current session which includes configuration and directory information.
- **frequency** (*Optional[str]*) – Variable's data frequency, if available.
- ****extra_facets** – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns
Path to the fixed file.

Return type
str or pathlib.Path

esmvalcore.preprocessor.**fix_metadata**(*cubes: Sequence[Cube]*, *short_name: str*, *project: str*, *dataset: str*, *mip: str*, *frequency: str | None = None*, *check_level: CheckLevels = CheckLevels.DEFAULT*, *session: Session | None = None*, ***extra_facets*) → CubeList

Fix cube metadata if fixes are required.

This method collects all the relevant fixes (including generic ones) for a given variable and applies them.

Parameters

- **cubes** (*Sequence[Cube]*) – Cubes to fix.
- **short_name** (*str*) – Variable's short name.
- **project** (*str*) – Project of the dataset.
- **dataset** (*str*) – Name of the dataset.
- **mip** (*str*) – Variable's MIP.
- **frequency** (*Optional[str]*) – Variable's data frequency, if available.
- **check_level** (CheckLevels) – Level of strictness of the checks.

  Deprecated since version 2.10.0: This option has been deprecated in ESMValCore version 2.10.0 and is scheduled for removal in version 2.12.0. Please use the functions *cmor_check_metadata()*, *cmor_check_data()*, or *cmor_check()* instead. This function will no longer perform CMOR checks. Fixes and CMOR checks have been clearly separated in ESMValCore version 2.10.0.
- **session** (*Optional[Session]*) – Current session which includes configuration and directory information.
- ****extra_facets** – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns
Fixed cubes.

Return type
iris.cube.CubeList

esmvalcore.preprocessor.**hourly_statistics**(*cube: Cube*, *hours: int*, *operator: str = 'mean'*, ***operator_kwargs*) → Cube

Compute hourly statistics.

Chunks time in x hours periods and computes statistics over them.

Parameters

- **cube** (*Cube*) – Input cube.
- **hours** (*int*) – Number of hours per period. Must be a divisor of 24, i.e., (1, 2, 3, 4, 6, 8, 12).
- **operator** (*str*) – The operation. Used to determine the iris.analysis.Aggregator object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the iris.analysis.Aggregator object defined by *operator*.

Returns
Hourly statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**linear_trend**(*cube*, *coordinate='time'*)

Calculate linear trend of data along a given coordinate.

The linear trend is defined as the slope of an ordinary linear regression.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.
- **coordinate** (*str, optional (default: 'time')*) – Dimensional coordinate over which the trend is calculated.

Returns
Trends.

Return type
iris.cube.Cube

Raises
iris.exceptions.CoordinateNotFoundError – The dimensional coordinate with the name `coordinate` is not found in `cube`.

esmvalcore.preprocessor.**linear_trend_stderr**(*cube*, *coordinate='time'*)

Calculate standard error of linear trend along a given coordinate.

This gives the standard error (not confidence intervals!) of the trend defined as the standard error of the estimated slope of an ordinary linear regression.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.
- **coordinate** (*str, optional (default: 'time')*) – Dimensional coordinate over which the standard error of the trend is calculated.

Returns
Standard errors of trends.

Return type
iris.cube.Cube

Raises
iris.exceptions.CoordinateNotFoundError – The dimensional coordinate with the name `coordinate` is not found in `cube`.

esmvalcore.preprocessor.**load**(*file: str | Path*, *ignore_warnings: list[dict] | None = None*) → CubeList

Load iris cubes from string or Path objects.

Parameters

- **file** (*str | Path*) – File to be loaded. Could be string or POSIX Path object.
- **ignore_warnings** (*list[dict] | None*) – Keyword arguments passed to `warnings.filterwarnings()` used to ignore warnings issued by `iris.load_raw()`. Each list element corresponds to one call to `warnings.filterwarnings()`.

Returns
Loaded cubes.

Return type
iris.cube.CubeList

Raises
ValueError – Cubes are empty.

esmvalcore.preprocessor.**mask_above_threshold**(*cube*, *threshold*)

Mask above a specific threshold value.

Takes a value 'threshold' and masks off anything that is above it in the cube data. Values equal to the threshold are not masked.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded.
- **threshold** (*float*) – threshold to be applied on input cube data.

Returns
thresholded cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**mask_below_threshold**(*cube*, *threshold*)

Mask below a specific threshold value.

Takes a value 'threshold' and masks off anything that is below it in the cube data. Values equal to the threshold are not masked.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **threshold** (*float*) – threshold to be applied on input cube data.

Returns
thresholded cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**mask_fillvalues**(*products*, *threshold_fraction*, *min_value=None*, *time_window=1*)

Compute and apply a multi-dataset fillvalues mask.

Construct the mask that fills a certain time window with missing values if the number of values in that specific window is less than a given fractional threshold. This function is the extension of _get_fillvalues_mask and performs the combination of missing values masks from each model (of multimodels) into a single fillvalues mask to be applied to each model.

Parameters

- **products** (*iris.cube.Cube*) – data products to be masked.
- **threshold_fraction** (*float*) – fractional threshold to be used as argument for Aggregator. Must be between 0 and 1.
- **min_value** (*float*) – minimum value threshold; default None If default, no thresholding applied so the full mask will be selected.
- **time_window** (*float*) – time window to compute missing data counts; default set to 1.

Returns
Masked iris cubes.

Return type
iris.cube.Cube

Raises
NotImplementedError – Implementation missing for data with higher dimensionality than 4.

esmvalcore.preprocessor.**mask_glaciated**(*cube*, *mask_out: str = 'glaciated'*)

Mask out glaciated areas.

It applies a Natural Earth mask. Note that for computational reasons only the 10 largest polygons are used for masking.

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **mask_out** (*str*) – "glaciated" to mask out glaciated areas

Returns
Returns the masked iris cube.

Return type
iris.cube.Cube

Raises
ValueError – Error raised if masking on irregular grids is attempted or if mask_out has a wrong value.

esmvalcore.preprocessor.**mask_inside_range**(*cube*, *minimum*, *maximum*)

Mask inside a specific threshold range.

Takes a MINIMUM and a MAXIMUM value for the range, and masks off anything that's between the two in the cube data.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns
thresholded cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**mask_landsea**(*cube*, *mask_out*)

Mask out either land mass or sea (oceans, seas and lakes).

It uses dedicated ancillary variables (sftlf or sftof) or, in their absence, it applies a Natural Earth mask (land or ocean contours). Note that the Natural Earth masks have different resolutions: 10m for land, and 50m for seas. These are more than enough for masking climate model data.

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked. If the cube has an `iris.coords.AncillaryVariable` with standard name `'land_area_fraction'` or `'sea_area_fraction'` that will be used. If both are present, only the 'land_area_fraction' will be used. If the ancillary variable is not available, the mask will be calculated from Natural Earth shapefiles.
- **mask_out** (*str*) – either "land" to mask out land mass or "sea" to mask out seas.

Returns
Returns the masked iris cube.

Return type
iris.cube.Cube

Raises
ValueError – Error raised if masking on irregular grids is attempted without an ancillary variable. Irregular grids are not currently supported for masking with Natural Earth shapefile masks.

esmvalcore.preprocessor.**mask_landseaice**(*cube*, *mask_out*)

Mask out either landsea (combined) or ice.

Function that masks out either landsea (land and seas) or ice (Antarctica, Greenland and some glaciers).

It uses dedicated ancillary variables (sftgif).

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked. It should have an iris.coords.AncillaryVariable with standard name 'land_ice_area_fraction'.
- **mask_out** (*str*) – either "landsea" to mask out landsea or "ice" to mask out ice.

Returns
Returns the masked iris cube with either land or ice masked out.

Return type
iris.cube.Cube

Raises
ValueError – Error raised if landsea-ice mask not found as an ancillary variable.

esmvalcore.preprocessor.**mask_multimodel**(*products*)

Apply common mask to all datasets (using logical OR).

Parameters
products (*iris.cube.CubeList or list of PreprocessorFile*) – Data products/cubes to be masked.

Returns
Masked data products/cubes.

Return type
iris.cube.CubeList or list of PreprocessorFile

Raises

- **ValueError** – Datasets have different shapes.
- **TypeError** – Invalid input data.

esmvalcore.preprocessor.**mask_outside_range**(*cube*, *minimum*, *maximum*)

Mask outside a specific threshold range.

Takes a MINIMUM and a MAXIMUM value for the range, and masks off anything that's outside the two in the cube data.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns
thresholded cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**meridional_statistics**(*cube: Cube*, *operator: str*, ***operator_kwargs*) → Cube

Compute meridional statistics.

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Meridional statistics cube.

Return type
iris.cube.Cube

Raises
ValueError – Error raised if computation on irregular grids is attempted. Zonal statistics not yet implemented for irregular grids.

esmvalcore.preprocessor.**monthly_statistics**(*cube: Cube*, *operator: str = 'mean'*, ***operator_kwargs*) → Cube

Compute monthly statistics.

Chunks time in monthly periods and computes statistics over them;

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Monthly statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**multi_model_statistics**(*products: set[PreprocessorFile] | Iterable[Cube]*, *span: str*, *statistics: list[str]*, *output_products=None*, *groupby: tuple | None = None*, *keep_input_datasets: bool = True*, *ignore_scalar_coords: bool = False*) → dict | set

Compute multi-model statistics.

This function computes multi-model statistics on a list of `products`, which can be instances of `Cube` or `PreprocessorFile`. The latter is used internally by ESMValCore to store workflow and provenance information, and this option should typically be ignored.

Cubes must have consistent shapes apart from a potential time dimension. There are two options to combine time coordinates of different lengths, see the `span` argument.

Uses the statistical operators in `iris.analysis`, including `mean`, `median`, `min`, `max`, and `std`. Percentiles are also supported and can be specified like `pXX.YY` (for percentile `XX.YY`; decimal part optional).

This function can handle cubes with differing metadata:

- Cubes with identical `name()` and `units` will get identical values for `standard_name`, `long_name`, and `var_name` (which will be arbitrarily set to the first encountered value if different cubes have different values for them).
- `attributes`: Differing attributes are deleted, see `iris.util.equalise_attributes()`.
- `cell_methods`: All cell methods are deleted prior to combining cubes.
- `cell_measures()`: All cell measures are deleted prior to combining cubes, see `esmvalcore.preprocessor.remove_fx_variables()`.
- `ancillary_variables()`: All ancillary variables are deleted prior to combining cubes, see `esmvalcore.preprocessor.remove_fx_variables()`.
- `coords()`: Exactly identical coordinates are preserved. For coordinates with equal `name()` and `units`, names are equalized, `attributes` deleted and `circular` is set to `False`. For all other coordinates, `long_name` is removed, `attributes` deleted and `circular` is set to `False`. Scalar coordinates can be removed if desired by the option `ignore_scalar_coords=True`. Please note that some special scalar coordinates which are expected to differ across cubes (ancillary coordinates for derived coordinates like *p0* and *ptop*) are always removed.

Notes

Some of the operators in `iris.analysis` require additional arguments. Except for percentiles, these operators are currently not supported.

Lazy operation is supported for all statistics, except `median`.

Parameters

- **products** (*set[PreprocessorFile] | Iterable[Cube]*) – Cubes (or products) over which the statistics will be computed.
- **span** (*str*) – Overlap or full; if overlap, statitstics are computed on common time- span; if full, statistics are computed on full time spans, ignoring missing data. This option is ignored if input cubes do not have time dimensions.
- **statistics** (*list[str]*) – Statistical metrics to be computed, e.g. [`mean`, `max`]. Allowed options are given in *this table*.
- **output_products** (*dict*) – For internal use only. A dict with statistics names as keys and preprocessorfiles as values. If products are passed as input, the statistics cubes will be assigned to these output products.
- **groupby** (*Optional[tuple]*) – Group products by a given tag or attribute, e.g., ('project', 'dataset', 'tag1'). This is ignored if `products` is a list of cubes.
- **keep_input_datasets** (*bool*) – If True, the output will include the input datasets. If False, only the computed statistics will be returned.
- **ignore_scalar_coords** (*bool*) – If True, remove any scalar coordinate in the input datasets before merging the input cubes into the multi-dataset cube. The resulting multi-dataset cube will have no scalar coordinates (the actual input datasets will remain unchanged). If False, scalar coordinates will remain in the input datasets, which might lead to merge conflicts in case the input datasets have different scalar coordinates.

Returns

A `dict` of cubes or `set` of *output_products* depending on the type of *products*.

Return type

dict | set

Raises
ValueError – If span is neither overlap nor full, or if input type is neither cubes nor products.

esmvalcore.preprocessor.**regrid**(*cube*, *target_grid*, *scheme*, *lat_offset=True*, *lon_offset=True*)

Perform horizontal regridding.

Note that the target grid can be a cube (Cube), path to a cube (str), a grid spec (str) in the form of *MxN*, or a dict specifying the target grid.

For the latter, the target_grid should be a dict with the following keys:

- start_longitude: longitude at the center of the first grid cell.
- end_longitude: longitude at the center of the last grid cell.
- step_longitude: constant longitude distance between grid cell centers.
- start_latitude: latitude at the center of the first grid cell.
- end_latitude: longitude at the center of the last grid cell.
- step_latitude: constant latitude distance between grid cell centers.

Parameters

- **cube** (Cube) – The source cube to be regridded.
- **target_grid** (*Cube or str or dict*) – The (location of a) cube that specifies the target or reference grid for the regridding operation. Alternatively, a string cell specification may be provided, of the form MxN, which specifies the extent of the cell, longitude by latitude (degrees) for a global, regular target grid. Alternatively, a dictionary with a regional target grid may be specified (see above).
- **scheme** (*str or dict*) – The regridding scheme to perform. If both source and target grid are structured (regular or irregular), can be one of the built-in schemes linear, linear_extrapolate, nearest, area_weighted, unstructured_nearest. Alternatively, a *dict* that specifies generic regridding (see below).
- **lat_offset** (*bool*) – Offset the grid centers of the latitude coordinate w.r.t. the pole by half a grid step. This argument is ignored if target_grid is a cube or file.
- **lon_offset** (*bool*) – Offset the grid centers of the longitude coordinate w.r.t. Greenwich meridian by half a grid step. This argument is ignored if target_grid is a cube or file.

Returns
Regridded cube.

Return type
Cube

See also:

extract_levels
Perform vertical regridding.

Notes

This preprocessor allows for the use of arbitrary Iris regridding schemes, that is anything that can be passed as a scheme to `iris.cube.Cube.regrid()` is possible. This enables the use of further parameters for existing schemes, as well as the use of more advanced schemes for example for unstructured meshes. To use this functionality, a dictionary must be passed for the scheme with a mandatory entry of `reference` in the form specified for the object reference of the entry point data model, i.e. `importable.module:object.attr`. This is used as a factory for the scheme. Any further entries in the dictionary are passed as keyword arguments to the factory.

For example, to use the familiar `iris.analysis.Linear` regridding scheme with a custom extrapolation mode, use

```
my_preprocessor:
  regrid:
    target: 1x1
    scheme:
      reference: iris.analysis:Linear
      extrapolation_mode: nanmask
```

To use the area weighted regridder available in `esmf_regrid.schemes.ESMFAreaWeighted` use

```
my_preprocessor:
  regrid:
    target: 1x1
    scheme:
      reference: esmf_regrid.schemes:ESMFAreaWeighted
```

esmvalcore.preprocessor.**regrid_time**(*cube: Cube*, *frequency: str*) → Cube

Align time axis for cubes so they can be subtracted.

Operations on time units, time points and auxiliary coordinates so that any cube from cubes can be subtracted from any other cube from cubes. Currently this function supports yearly (frequency=yr), monthly (frequency=mon), daily (frequency=day), 6-hourly (frequency=6hr), 3-hourly (frequency=3hr) and hourly (frequency=1hr) data time frequencies.

Parameters

- **cube** (*Cube*) – Input cube.
- **frequency** (*str*) – Data frequency: *mon*, *day*, *1hr*, *3hr* or *6hr*.

Returns
Cube with converted time axis and units.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**remove_supplementary_variables**(*cube: Cube*)

Remove supplementary variables.

Strip cell measures or ancillary variables from the cube containing the data.

Parameters
cube (*iris.cube.Cube*) – Iris cube with data and cell measures or ancillary variables.

Returns
Cube without cell measures or ancillary variables.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**resample_hours**(*cube: Cube*, *interval: int*, *offset: int = 0*) → Cube

Convert x-hourly data to y-hourly by eliminating extra timesteps.

Convert x-hourly data to y-hourly (y > x) by eliminating the extra timesteps. This is intended to be used only with instantaneous values.

For example:

- resample_hours(cube, interval=6): Six-hourly intervals at 0:00, 6:00, 12:00, 18:00.
- resample_hours(cube, interval=6, offset=3): Six-hourly intervals at 3:00, 9:00, 15:00, 21:00.
- resample_hours(cube, interval=12, offset=6): Twelve-hourly intervals at 6:00, 18:00.

Parameters

- **cube** (*Cube*) – Input cube.
- **interval** (*int*) – The period (hours) of the desired data.
- **offset** (*optional*) – The firs hour (hours) of the desired data.

Returns
Cube with the new frequency.

Return type
iris.cube.Cube

Raises
ValueError: – The specified frequency is not a divisor of 24.

esmvalcore.preprocessor.**resample_time**(*cube: Cube*, *month: int | None = None*, *day: int | None = None*, *hour: int | None = None*) → Cube

Change frequency of data by resampling it.

Converts data from one frequency to another by extracting the timesteps that match the provided month, day and/or hour. This is meant to be used with instantaneous values when computing statistics is not desired.

For example:

- resample_time(cube, hour=6): Daily values taken at 6:00.
- resample_time(cube, day=15, hour=6): Monthly values taken at 15th 6:00.
- resample_time(cube, month=6): Yearly values, taking in June
- resample_time(cube, month=6, day=1): Yearly values, taking 1st June

The condition must yield only one value per interval: the last two samples above will produce yearly data, but the first one is meant to be used to sample from monthly output and the second one will work better with daily.

Parameters

- **cube** (*Cube*) – Input cube.
- **month** (*optional*) – Month to extract.
- **day** (*optional*) – Day to extract.
- **hour** (*optional*) – Hour to extract.

Returns
Cube with the new frequency.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**rolling_window_statistics**(*cube: Cube*, *coordinate: str*, *operator: str*, *window_length: int*, ***operator_kwargs*)

Compute rolling-window statistics over a coordinate.

Parameters

- **cube** (*Cube*) – Input cube.
- **coordinate** (*str*) – Coordinate over which the rolling-window statistics is calculated.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.
- **window_length** (*int*) – Size of the window to use.

Returns
Rolling-window statistics cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**save**(*cubes*, *filename*, *optimize_access=''*, *compress=False*, *alias=''*, ***kwargs*)

Save iris cubes to file.

Parameters

- **cubes** (*iterable of iris.cube.Cube*) – Data cubes to be saved
- **filename** (*str*) – Name of target file
- **optimize_access** (*str*) – Set internal NetCDF chunking to favour a reading scheme

  Values can be map or timeseries, which improve performance when reading the file one map or time series at a time. Users can also provide a coordinate or a list of coordinates. In that case the better performance will be avhieved by loading all the values in that coordinate at a time
- **compress** (*bool, optional*) – Use NetCDF internal compression.
- **alias** (*str, optional*) – Var name to use when saving instead of the one in the cube.

Returns
filename

Return type
str

Raises
ValueError – cubes is empty.

esmvalcore.preprocessor.**seasonal_statistics**(*cube: Cube*, *operator: str = 'mean'*, *seasons: Iterable[str] = ('DJF', 'MAM', 'JJA', 'SON')*, ***operator_kwargs*) → Cube

Compute seasonal statistics.

Chunks time seasons and computes statistics over them.

Parameters

- **cube** (*Cube*) – Input cube.

- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- **seasons** (*Iterable[str]*) – Seasons to build. Available: ('DJF', 'MAM', 'JJA', SON') (default) and all sequentially correct combinations holding every month of a year: e.g. ('JJAS','ONDJFMAM'), or less in case of prior season extraction.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Seasonal statistic cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**timeseries_filter**(*cube: Cube*, *window: int*, *span: int*, *filter_type: str = 'lowpass'*, *filter_stats: str = 'sum'*, ***operator_kwargs*) → Cube

Apply a timeseries filter.

Method borrowed from iris example

Apply each filter using the rolling_window method used with the weights keyword argument. A weighted sum is required because the magnitude of the weights are just as important as their relative sizes.

See also the iris rolling window `iris.cube.Cube.rolling_window`.

Parameters

- **cube** (*Cube*) – Input cube.
- **window** (*int*) – The length of the filter window (in units of cube time coordinate).
- **span** (*int*) – Number of months/days (depending on data frequency) on which weights should be computed e.g. 2-yearly: span = 24 (2 x 12 months). Span should have same units as cube time coordinate.
- **filter_type** (*optional*) – Type of filter to be applied; default 'lowpass'. Available types: 'lowpass'.
- **filter_stats** (*optional*) – Type of statistic to aggregate on the rolling window; default: *sum*. Used to determine the `iris.analysis.Aggregator` object used for aggregation. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *filter_stats*.

Returns
Cube time-filtered using 'rolling_window'.

Return type
iris.cube.Cube

Raises

- **iris.exceptions.CoordinateNotFoundError:** – Cube does not have time coordinate.
- **NotImplementedError:** – *filter_type* is not implemented.

esmvalcore.preprocessor.**volume_statistics**(*cube: Cube*, *operator: str*, ***operator_kwargs*) → Cube

Apply a statistical operation over a volume.

The volume average is weighted according to the cell volume.

Parameters

- **cube** (*Cube*) – Input cube. The input cube should have a `iris.coords.CellMeasure` named `'ocean_volume'`, unless it has regular 1D latitude and longitude coordinates so the cell volumes can be computed by using `iris.analysis.cartography.area_weights()` to compute the cell areas and multiplying those by the cell thickness, computed from the bounds of the vertical coordinate.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Currently, only *mean* is allowed.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Collapsed cube.

Return type
iris.cube.Cube

esmvalcore.preprocessor.**weighting_landsea_fraction**(*cube*, *area_type*)

Weight fields using land or sea fraction.

This preprocessor function weights a field with its corresponding land or sea area fraction (value between 0 and 1). The application of this is important for most carbon cycle variables (and other land-surface outputs), which are e.g. reported in units of *kgC m-2*. This actually refers to 'per square meter of land/sea' and NOT 'per square meter of gridbox'. So in order to integrate these globally or regionally one has to both area-weight the quantity but also weight by the land/sea fraction.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to be weighted. It should have an `iris.coords.AncillaryVariable` with standard name `'land_area_fraction'` or `'sea_area_fraction'`. If both are present, only the `'land_area_fraction'` will be used.
- **area_type** (*str*) – Use land (`'land'`) or sea (`'sea'`) fraction for weighting.

Returns
Land/sea fraction weighted cube.

Return type
iris.cube.Cube

Raises

- **TypeError** – `area_type` is not `'land'` or `'sea'`.
- **ValueError** – Land/sea fraction variables `sftlf` or `sftof` not found.

esmvalcore.preprocessor.**zonal_statistics**(*cube: Cube*, *operator: str*, ***operator_kwargs*) → Cube

Compute zonal statistics.

Parameters

- **cube** (*Cube*) – Input cube.
- **operator** (*str*) – The operation. Used to determine the `iris.analysis.Aggregator` object used to calculate the statistics. Allowed options are given in *this table*.
- ****operator_kwargs** – Optional keyword arguments for the `iris.analysis.Aggregator` object defined by *operator*.

Returns
Zonal statistics cube.

Return type
iris.cube.Cube

Raises
`ValueError` – Error raised if computation on irregular grids is attempted. Zonal statistics not yet implemented for irregular grids.

CHAPTER

THIRTYSEVEN

TYPE HINTS

Type aliases for providing type hints.

esmvalcore.typing.**FacetValue**(**args*, ***kwargs*)

Type describing a single facet.

alias of Union[str, Sequence[str], Number]

esmvalcore.typing.**Facets**(**args*, ***kwargs*)

Type describing a collection of facets.

alias of Dict[str, Union[str, Sequence[str], Number]]

esmvalcore.typing.**NetCDFAttr**(**args*, ***kwargs*)

Type describing netCDF attributes.

NetCDF attributes can be strings, numbers or sequences.

alias of Union[str, Number, Iterable]

CHAPTER

THIRTYEIGHT

EXPERIMENTAL API

This page describes the new ESMValCore API. The experimental API module is available in the submodule `esmvalcore.experimental`. The API is under development, so use at your own risk!

38.1 Recipes

This section describes the *recipe* submodule of the API (`esmvalcore.experimental`).

38.1.1 Recipe metadata

Recipe is a class that holds metadata from a recipe.

```
>>> Recipe('path/to/recipe_python.yml')
recipe = Recipe('Recipe Python')
```

Printing the recipe will give a nice overview of the recipe:

```
>>> print(recipe)
## Recipe python

Example recipe that plots a map and timeseries of temperature.

### Authors
 - Bouwe Andela (NLeSC, Netherlands; https://orcid.org/0000-0001-9005-8940)
 - Mattia Righi (DLR, Germany; https://orcid.org/0000-0003-3827-5950)

### Maintainers
 - Manuel Schlund (DLR, Germany; https://orcid.org/0000-0001-5251-0158)

### Projects
 - DLR project ESMVal
 - Copernicus Climate Change Service 34a Lot 2 (MAGIC) project

### References
 - Please acknowledge the project(s).
```

38.1.2 Running a recipe

To run the recipe, call the *run()* method.

```
>>> output = recipe.run()
<log messages>
```

By default, a new `Session` is automatically created, so that data are never overwritten. Data are stored in the `esmvaltool_output` directory specified in the config. Sessions can also be explicitly specified.

```
>>> from esmvalcore.experimental import CFG
>>> session = CFG.start_session('my_session')
>>> output = recipe.run(session)
<log messages>
```

run() returns an dictionary of objects that can be used to inspect the output of the recipe. The output is an instance of *ImageFile* or *DataFile* depending on its type.

For working with recipe output, see: *Recipe output*.

38.1.3 Running a single diagnostic or preprocessor task

The python example recipe contains 5 tasks:

Preprocessors:

- `timeseries/tas_amsterdam`
- `timeseries/script1`
- `map/tas`

Diagnostics:

- `timeseries/tas_global`
- `map/script1`

To run a single diagnostic or preprocessor, the name of the task can be passed as an argument to *run()*. If a diagnostic is passed, all ancestors will automatically be run too.

```
>>> output = recipe.run('map/script1')
>>> output
map/script1:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
  ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
  ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')
```

It is also possible to run a single preprocessor task:

```
>>> output = recipe.run('map/tas')
>>> output
map/tas:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
```

38.1.4 API reference

Recipe metadata.

Classes:

Recipe(path)	API wrapper for the esmvalcore Recipe object.

class esmvalcore.experimental.recipe.**Recipe**(*path: PathLike*)

Bases: object

API wrapper for the esmvalcore Recipe object.

This class can be used to inspect and run the recipe.

Parameters
path (*pathlike*) – Path to the recipe.

Attributes:

data	Return dictionary representation of the recipe.
name	Return the name of the recipe.

Methods:

get_output()	Get output from recipe.
render([template])	Render output as html.
run([task, session])	Run the recipe.

property data: dict

Return dictionary representation of the recipe.

get_output() → *RecipeOutput*

Get output from recipe.

Returns
output – Returns output of the recipe as instances of `OutputFile` grouped by diagnostic task.

Return type
dict

property name

Return the name of the recipe.

render(*template=None*)

Render output as html.

template
[`Template`] An instance of `jinja2.Template` can be passed to customize the output.

run(*task: str | None = None*, *session: Session | None = None*)

Run the recipe.

This function loads the recipe into the ESMValCore recipe format and runs it.

Parameters

- **task** (*str*) – Specify the name of the diagnostic or preprocessor to run a single task.
- **session** (Session, optional) – Defines the config parameters and location where the recipe output will be stored. If None, a new session will be started automatically.

Returns

output – Returns output of the recipe as instances of OutputItem grouped by diagnostic task.

Return type

dict

38.2 Recipe output

This section describes the *recipe_output* submodule of the API (esmvalcore.experimental).

After running a recipe, output is returned by the *run()* method. Alternatively, it can be retrieved using the *get_output()* method.

```
>>> recipe_output = recipe.get_output()
```

recipe_output is a mapping of the individual tasks and their output filenames (data and image files) with a set of attributes describing the data.

```
>>> recipe_output
timeseries/script1:
  DataFile('tas_amsterdam_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.nc')
  DataFile('tas_amsterdam_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.nc')
  DataFile('tas_amsterdam_MultiModelMean_Amon_tas_1850-2000.nc')
  DataFile('tas_global_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.nc')
  DataFile('tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.nc')
  ImageFile('tas_amsterdam_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.png')
  ImageFile('tas_amsterdam_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.png')
  ImageFile('tas_amsterdam_MultiModelMean_Amon_tas_1850-2000.png')
  ImageFile('tas_global_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.png')
  ImageFile('tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.png')

map/script1:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
  ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
  ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')
```

Output is grouped by the task that produced them. They can be accessed like a dictionary.

```
>>> task_output = recipe_output['map/script1']
>>> task_output
map/script1:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
  ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
  ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')
```

The task output has a list of files associated with them, usually image (.png) or data files (.nc). To get a list of all files, use files().

```
>>> print(task_output.files)
(DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc'),
..., ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png'))
```

It is also possible to select the image (*image_files()*) files or data files (*data_files()*) only.

```
>>> for image_file in task_output.image_files:
>>>     print(image_file)
ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')

>>> for data_file in task_output.data_files:
>>>     print(data_file)
DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
```

38.2.1 Working with output files

Output comes in two kinds, *DataFile* corresponds to data files in `.nc` format and *ImageFile* corresponds to plots in `.png` format (see below). Both object are derived from the same base class (*OutputFile*) and therefore share most of the functionality.

For example, author information can be accessed as instances of *Contributor* via

```
>>> output_file = task_output[0]
>>> output_file.authors
(Contributor('Andela, Bouwe', institute='NLeSC, Netherlands', orcid='https://orcid.org/
↪0000-0001-9005-8940'),
 Contributor('Righi, Mattia', institute='DLR, Germany', orcid='https://orcid.org/0000-
↪0003-3827-5950'))
```

And associated references as instances of *Reference* via

```
>>> output_file.references
(Reference('acknow_project'),)
```

OutputFile also knows about associated files

```
>>> data_file.citation_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_citation.
↪bibtex')
>>> data_file.data_citation_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_data_citation_
↪info.txt')
>>> data_file.provenance_svg_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_provenance.svg
↪')
>>> data_file.provenance_xml_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_provenance.xml
↪')
```

38.2.2 Working with image files

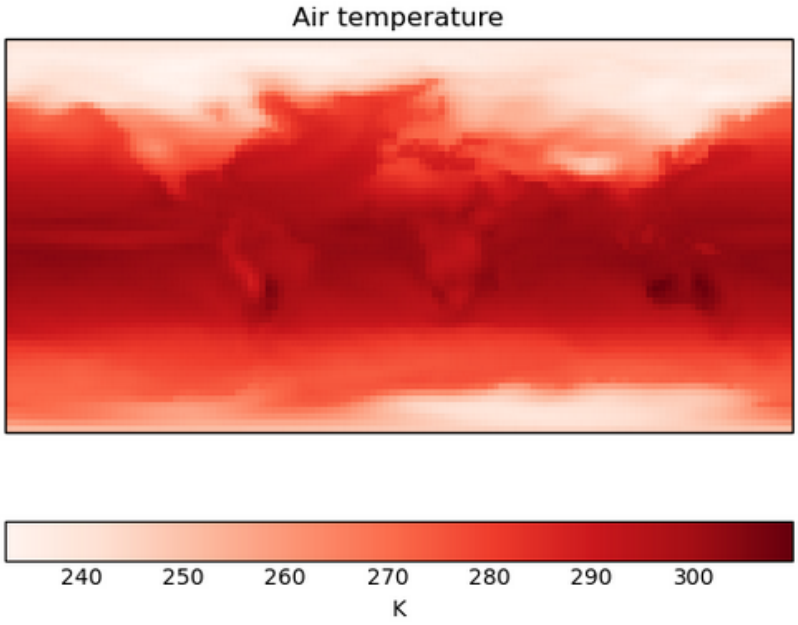
Image output uses IPython magic to plot themselves in a notebook environment.

```
>>> image_file = recipe_output['map/script1'].image_files[0]
>>> image_file
```

For example:

```
In [30]: image_file = recipe_output['map/script1'].image_files[0]
         image_file
```

Out[30]: Average Near-Surface Air Temperature between 2000 and 2000 according to BCC-ESM1.



Using `IPython.display`, it is possible to show all image files.

```
>>> from IPython.display import display
>>>
>>> task = recipe_output['map/script1']
>>> for image_file in task.image_files:
>>>     display(image_file)
```

38.2.3 Working with data files

Data files can be easily loaded using `xarray`:

```
>>> data_file = recipe_output['timeseries/script1'].data_files[0]
>>> data = data_file.load_xarray()
>>> type(data)
xarray.core.dataset.Dataset
```

Or `iris`:

```
>>> cube = data_file.load_iris()
>>> type(cube)
iris.cube.CubeList
```

38.2.4 API reference

API for handing recipe output.

Classes:

DataFile(path[, attributes])	Container for data output.
DiagnosticOutput(name, task_output[, title, ...])	Container for diagnostic output.
ImageFile(path[, attributes])	Container for image output.
OutputFile(path[, attributes])	Base container for recipe output files.
RecipeOutput(task_output[, session, info])	Container for recipe output.
TaskOutput(name, files)	Container for task output.

class esmvalcore.experimental.recipe_output.**DataFile**(*path: str*, *attributes: dict | None = None*)

Bases: *OutputFile*

Container for data output.

Attributes:

authors	List of recipe authors.
caption	Return the caption of the file (fallback to path).
citation_file	Return path of citation file (bibtex format).
data_citation_file	Return path of data citation info (txt format).
kind	
provenance_xml_file	Return path of provenance file (xml format).
references	List of project references.

Methods:

create(path[, attributes])	Construct new instances of OutputFile.
load_iris()	Load data using iris.
load_xarray()	Load data using xarray.

Parameters

- **path** (*str*) –
- **attributes** (*dict | None*) –

property authors: tuple

List of recipe authors.

property caption: str

Return the caption of the file (fallback to path).

property citation_file

Return path of citation file (bibtex format).

classmethod create(*path: str*, *attributes: dict | None = None*) → *OutputFile*

Construct new instances of OutputFile.

Chooses a derived class if suitable.

Parameters

- **path** (*str*) –
- **attributes** (*dict | None*) –

Return type
OutputFile

property data_citation_file

Return path of data citation info (txt format).

kind: str | None = 'data'

load_iris()

Load data using iris.

load_xarray()

Load data using xarray.

property provenance_xml_file

Return path of provenance file (xml format).

property references: tuple

List of project references.

class esmvalcore.experimental.recipe_output.**DiagnosticOutput**(*name*, *task_output*, *title=None*, *description=None*)

Bases: object

Container for diagnostic output.

Parameters

- **name** (*str*) – Name of the diagnostic
- **title** (*str*) – Title of the diagnostic
- **description** (*str*) – Description of the diagnostic
- **task_output** (list of *TaskOutput*) – List of task output.

class esmvalcore.experimental.recipe_output.**ImageFile**(*path: str*, *attributes: dict | None = None*)

Bases: *OutputFile*

Container for image output.

Attributes:

authors	List of recipe authors.
caption	Return the caption of the file (fallback to path).
citation_file	Return path of citation file (bibtex format).
data_citation_file	Return path of data citation info (txt format).
kind	
provenance_xml_file	Return path of provenance file (xml format).
references	List of project references.

Methods:

create(path[, attributes])	Construct new instances of OutputFile.
to_base64()	Encode image as base64 to embed in a Jupyter notebook.

Parameters

- **path** (*str*) –
- **attributes** (*dict | None*) –

property **authors**: tuple

List of recipe authors.

property **caption**: str

Return the caption of the file (fallback to path).

property **citation_file**

Return path of citation file (bibtex format).

classmethod **create**(*path: str*, *attributes: dict | None = None*) → *OutputFile*

Construct new instances of OutputFile.

Chooses a derived class if suitable.

Parameters

- **path** (*str*) –
- **attributes** (*dict | None*) –

Return type
OutputFile

property **data_citation_file**

Return path of data citation info (txt format).

kind: str | None = 'image'

property provenance_xml_file

Return path of provenance file (xml format).

property references: tuple

List of project references.

to_base64() → str

Encode image as base64 to embed in a Jupyter notebook.

Return type
str

class esmvalcore.experimental.recipe_output.OutputFile(*path: str*, *attributes: dict | None = None*)

Bases: object

Base container for recipe output files.

Use *OutputFile.create(path='<path>', attributes=attributes)* to initialize a suitable subclass.

Parameters

- **path** (*str*) – Name of output file
- **attributes** (*dict*) – Attributes corresponding to the recipe output

Attributes:

authors	List of recipe authors.
caption	Return the caption of the file (fallback to path).
citation_file	Return path of citation file (bibtex format).
data_citation_file	Return path of data citation info (txt format).
kind	
provenance_xml_file	Return path of provenance file (xml format).
references	List of project references.

Methods:

create(path[, attributes])	Construct new instances of OutputFile.

property authors: tuple

List of recipe authors.

property caption: str

Return the caption of the file (fallback to path).

property citation_file

Return path of citation file (bibtex format).

classmethod create(*path: str*, *attributes: dict | None = None*) → *OutputFile*

Construct new instances of OutputFile.

Chooses a derived class if suitable.

Parameters

- **path** (*str*) –
- **attributes** (*dict | None*) –

Return type
OutputFile

property data_citation_file

Return path of data citation info (txt format).

kind: str | None = None

property provenance_xml_file

Return path of provenance file (xml format).

property references: tuple

List of project references.

class esmvalcore.experimental.recipe_output.**RecipeOutput**(*task_output: dict*, *session=None*, *info=None*)

Bases: Mapping

Container for recipe output.

Parameters
task_output (*dict*) – Dictionary with recipe output grouped by task name. Each task value is a mapping of the filenames with the product attributes.

diagnostics

Dictionary with recipe output grouped by diagnostic.

Type
dict

info

The recipe used to create the output.

Type
RecipeInfo

session

The session used to run the recipe.

Type
esmvalcore.config.Session

Attributes:

FILTER_ATTRS

Methods:

from_core_recipe_output(recipe_output)	Construct instance from *_recipe.Recipe* output.
get(k[,d])	
items()	
keys()	
read_main_log()	Read log file.
read_main_log_debug()	Read debug log file.
render([template])	Render output as html.
values()	
write_html()	Write output summary to html document.

FILTER_ATTRS: list = ['realms', 'plot_type', 'plot_types', 'long_names']

classmethod from_core_recipe_output(*recipe_output: dict*)

Construct instance from *_recipe.Recipe* output.

The core recipe format is not directly compatible with the API. This constructor converts the raw recipe dict to `RecipeInfo`

Parameters
recipe_output (*dict*) – Output from *_recipe.Recipe.get_product_output*

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

read_main_log() → str

Read log file.

Return type
str

read_main_log_debug() → str

Read debug log file.

Return type
str

render(*template=None*)

Render output as html.

template
[`Template`] An instance of `jinja2.Template` can be passed to customize the output.

values() → an object providing a view on D's values

write_html()

Write output summary to html document.

A html file *index.html* gets written to the session directory.

class esmvalcore.experimental.recipe_output.**TaskOutput**(*name: str*, *files: dict*)

Bases: object

Container for task output.

Parameters

- **name** (*str*) – Name of the task
- **files** (*dict*) – Mapping of the filenames with the associated attributes.

Attributes:

data_files	Return a tuple of data objects.
image_files	Return a tuple of image objects.

Methods:

from_task(task)	Create an instance of *TaskOutput* from a Task.

property data_files: tuple

Return a tuple of data objects.

classmethod from_task(*task*) → *TaskOutput*

Create an instance of *TaskOutput* from a Task.

Where task is an instance of *esmvalcore._task.BaseTask*.

Return type
TaskOutput

property image_files: tuple

Return a tuple of image objects.

38.3 Recipe Metadata

This section describes the *recipe_metadata* submodule of the API (`esmvalcore.experimental`).

38.3.1 API reference

API for recipe metadata.

Classes:

Contributor(name, institute[, orcid])	Contains contributor (author or maintainer) information.
Project(project)	Use this class to acknowledge a project associated with the recipe.
Reference(filename)	Parse reference information from bibtex entries.

Exceptions:

RenderError	Error during rendering of object.

class esmvalcore.experimental.recipe_metadata.**Contributor**(*name: str*, *institute: str*, *orcid: str | None = None*)

Bases: object

Contains contributor (author or maintainer) information.

Parameters

- **name** (*str*) – Name of the author, i.e. 'John Doe'
- **institute** (*str*) – Name of the institute
- **orcid** (*str, optional*) – ORCID url

Methods:

from_dict(attributes)	Return an instance of Contributor from a dictionary.
from_tag(tag)	Return an instance of Contributor from a tag (TAGS).

classmethod from_dict(*attributes*)

Return an instance of Contributor from a dictionary.

Parameters

attributes (*dict*) – Dictionary containing name / institute [/ orcid].

classmethod from_tag(*tag: str*) → *Contributor*

Return an instance of Contributor from a tag (TAGS).

Parameters

tag (*str*) – The contributor tags are defined in the authors section in config-references.yml.

Return type

Contributor

class esmvalcore.experimental.recipe_metadata.**Project**(*project: str*)

Bases: object

Use this class to acknowledge a project associated with the recipe.

Parameters

project (*str*) – The project title.

Methods:

from_tag(tag)	Return an instance of Project from a tag (TAGS).

classmethod from_tag(*tag: str*) → *Project*

Return an instance of Project from a tag (TAGS).

Parameters

tag (*str*) – The project tags are defined in config-references.yml.

Return type

Project

class esmvalcore.experimental.recipe_metadata.**Reference**(*filename: str*)

Bases: object

Parse reference information from bibtex entries.

Parameters
filename (*str*) – Name of the bibtex file.

Raises
NotImplementedError – If the bibtex file contains more than 1 entry.

Methods:

from_tag(tag)	Return an instance of Reference from a bibtex tag.
render([renderer])	Render the reference.

classmethod from_tag(*tag: str*) → *Reference*

Return an instance of Reference from a bibtex tag.

Parameters
tag (*str*) – The bibtex tags resolved as `esmvaltool/references/{tag}.bibtex` or the corresponding directory as defined by the diagnostics path.

Return type
Reference

render(*renderer: str = 'html'*) → str

Render the reference.

Parameters
renderer (*str*) – Choose the renderer for the string representation. Must be one of: 'plain-text', 'markdown', 'html', 'latex'

Returns
Rendered reference

Return type
str

exception esmvalcore.experimental.recipe_metadata.**RenderError**

Bases: BaseException

Error during rendering of object.

add_note()

Exception.add_note(note) – add a note to the exception

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

38.4 Utils

This section describes the *utils* submodule of the API (`esmvalcore.experimental`).

38.4.1 Finding recipes

One of the first thing we may want to do, is to simply get one of the recipes available in `ESMValTool`

If you already know which recipe you want to load, call *get_recipe()*.

```
from esmvalcore.experimental import get_recipe
>>> get_recipe('examples/recipe_python')
Recipe('Recipe python')
```

Call the *get_all_recipes()* function to get a list of all available recipes.

```
>>> from esmvalcore.experimental import get_all_recipes
>>> recipes = get_all_recipes()
>>> recipes
[Recipe('Recipe perfmetrics cmip5 4cds'),
 Recipe('Recipe martin18grl'),
 ...
 Recipe('Recipe wflow'),
 Recipe('Recipe pcrglobwb')]
```

To search for a specific recipe, you can use the *find()* method. This takes a search query that looks through the recipe metadata and returns any matches. The query can be a regex pattern, so you can make it as complex as you like.

```
>>> results = recipes.find('climwip')
[Recipe('Recipe climwip')]
```

The recipes are loaded in a *Recipe* object, which knows about the documentation, authors, project, and related references of the recipe. It resolves all the tags, so that it knows which institute an author belongs to and which references are associated with the recipe.

This means you can search for something like this:

```
>>> recipes.find('Geophysical Research Letters')
[Recipe('Recipe martin18grl'),
 Recipe('Recipe climwip'),
 Recipe('Recipe ecs constraints'),
 Recipe('Recipe ecs scatter'),
 Recipe('Recipe ecs'),
 Recipe('Recipe seaice')]
```

38.4.2 API reference

ESMValCore utilities.

Classes:

RecipeList([iterable])	Container for recipes.

Functions:

get_all_recipes([subdir])	Return a list of all available recipes.
get_recipe(name)	Get a recipe by its name.

class esmvalcore.experimental.utils.**RecipeList**(*iterable=()*, */*)

Container for recipes.

Methods:

find(query)	Search for recipes matching the search query or pattern.

find(*query: Pattern[str]*)

Search for recipes matching the search query or pattern.

Searches in the description, authors and project information fields. All matches are returned.

Parameters
query (*str, Pattern*) – String to search for, e.g. `find_recipes('righi')` will return all matching that author. Can be a *regex* pattern.

Returns
List of recipes matching the search query.

Return type
RecipeList

esmvalcore.experimental.utils.**get_all_recipes**(*subdir: str | None = None*) → list

Return a list of all available recipes.

Parameters
subdir (*str*) – Sub-directory of the `DIAGNOSTICS.path` to look for recipes, e.g. `get_all_recipes(subdir='examples')`.

Returns
List of available recipes

Return type
RecipeList

esmvalcore.experimental.utils.**get_recipe**(*name: PathLike | str*) → *Recipe*

Get a recipe by its name.

The function looks first in the local directory, and second in the repository defined by the diagnostic path. The recipe name can be specified with or without extension. The first match will be returned.

Parameters
name (*str, pathlike*) – Name of the recipe file, i.e. `examples/recipe_python.yml`

Returns
Instance of `Recipe` which can be used to inspect and run the recipe.

Return type
Recipe

Raises
`FileNotFoundError` – If the name cannot be resolved to a recipe file.

Part VIII

Changelog

CHAPTER

THIRTYNINE

V2.10.0

Highlights

- All statistics preprocessors support the same operators and have a common *documentation*. In addition, arbitrary keyword arguments for the statistical operation can be directly given to the preprocessor.
- The output webpage generated by the tool now looks better and provides methods to select and filter the output.
- Improved computational efficiency:
  - Automatic rechunking between preprocessor steps to keep the graph size smaller and the chunk size optimal.
  - Reduce the size of the dask graph created by *esmvalcore.preprocessor.anomalies()*.
  - Preprocessors *esmvalcore.preprocessor.mask_above_threshold()*, *esmvalcore.preprocessor.mask_below_threshold()*, *esmvalcore.preprocessor.mask_inside_range()*, *esmvalcore.preprocessor.mask_outside_range()* are now lazy.
  - Lazy coordinates bounds are no longer loaded into memory by the CMOR checks and fixes.

This release includes

39.1 Backwards incompatible changes

- Remove the deprecated option `use_legacy_supplementaries` (#2202) Bouwe Andela
  - The recommended upgrade procedure is to remove `use_legacy_supplementaries` from config-user.yml (if it was there) and remove any mention of `fx_variables` from the recipe. If automatically defining the required supplementary variables does not work, define them in the variable or (`additional_`) `datasets` section as described in *Defining supplementary variables (ancillary variables and cell measures)*.
- Use smarter (units-aware) weights (#2139) Manuel Schlund
  - Some preprocessors handle units better. For details, see the pull request.
- Removed deprecated configuration option `offline` (#2213) Manuel Schlund
  - In *v2.8.0*, we replaced the old `offline` configuration option. From this version on, it stops working. Please refer to *v2.8.0* for upgrade instructions.
- Fix issue with CORDEX datasets requiring different dataset tags for downloads and fixes (#2066) Joakim Löw
  - Due to the different facets for CORDEX datasets, there was an inconsistency in the fixing mechanism. This change requires changes to existing recipes that use CORDEX datasets. Please refer to the pull request for detailed update instructions.
- For the following changes, no user change is necessary
  - Remove deprecated way of calling *read_cmor_tables()* (#2201) Bouwe Andela

  - Remove deprecated callback argument from preprocessor `load` function (#2207) Bouwe Andela
  - Remove deprecated preprocessor function *cleanup* (#2215) Bouwe Andela

39.2 Deprecations

- Clearly separate fixes and CMOR checks (#2157) Manuel Schlund
- Added new operators for statistics preprocessor (e.g., `'percentile'`) and allowed passing additional arguments (#2191) Manuel Schlund
  - This harmonizes the operators for all statistics preprocessors. From this version, the new names can be used; the old arguments will stop working from version 2.12.0. Please refer to *Statistical preprocessors* for a detailed description.

39.3 Bug fixes

- Re-add correctly region-extracted cell measures and ancillary variables after *extract_region* (#2166) Valeriu Predoi, Manuel Schlund
- Fix sorting of datasets
  - Fix sorting of ensemble members in *datasets_to_recipe()* (#2095) Bouwe Andela
  - Fix a problem with sorting datasets that have a mix of facet types (#2238) Bouwe Andela
  - Avoid a crash if dataset has supplementary variables (#2198) Bouwe Andela

39.4 CMOR standard

- ERA5 on-the-fly CMORizer: changed sign of variables `evspsbl` and `evspsblpot` (#2115) katjaweigel
- Add `ch4` surface custom cmor table entry (#2168) Birgit Hassler
- Add CMIP3 institutes names used at NCI (#2152) Romain Beucher
- Added *get_time_bounds()* and *get_next_month()* to public API (#2214) Manuel Schlund
- Improve concatenation checks
  - Relax concatenation checks for `--check_level=relax` and `--check_level=ignore` (#2144) sloosvel
  - Fix `concatenate` preprocessor function (#2240) Bouwe Andela
  - Fix time overlap handling in concatenation (#2247) Klaus Zimmermann

39.5 Computational performance improvements

- Make *Minimum, maximum and interval masking* preprocessors lazy (#2169) Jörg Benke
  - Restored usage of numpy in *_mask_with_shp* (#2209) Jörg Benke
- Do not realize lazy coordinate bounds in CMOR check (#2146) sloosvel
- Rechunk between preprocessor steps (#2205) Bouwe Andela
- Reduce the size of the dask graph created by the `anomalies` preprocessor function (#2200) Bouwe Andela

39.6 Documentation

- Add reference to release v2.9.0 in the changelog (#2130) Rémi Kazeroni
- Add merge instructions to release instructions (#2131) Klaus Zimmermann
- Update *mamba* before building environment during Readthedocs build (#2149) Valeriu Predoi
- Ensure compatible zstandard and zstd versions for .conda support (#2204) Klaus Zimmermann
- Remove outdated documentation (#2210) Bouwe Andela
- Remove meercode badge from README because their API is broken (#2224) Valeriu Predoi
- Correct usage help text of version command (#2232) James Frost
- Add `navigation_with_keys: False` to `html_theme_options` in Readthedocs `conf.py` (#2245) Valeriu Predoi
- Replace squarey badge with roundy shield for Anaconda sticker in README (#2233, #2260) Valeriu Predoi

39.7 Fixes for datasets

- Updated doc about fixes and added type hints to fix functions (#2160) Manuel Schlund

39.8 Installation

- Clean-up how pins are written in conda environment file (#2125) Valeriu Predoi
- Use importlib.metadata instead of deprecated pkg_resources (#2096) Bouwe Andela
- Pin shapely to >=2.0 (#2075) Valeriu Predoi
- Pin Python to <3.12 in conda environment (#2272) Bouwe Andela

39.9 Preprocessor

- Improve preprocessor output sorting code (#2111) Bouwe Andela
- Preprocess datasets in the same order as they are listed in the recipe (#2103) Bouwe Andela

39.10 Automatic testing

- [Github Actions] Compress all bash shell setters into one default option per workflow (#2126) Valeriu Predoi
- [Github Actions] Fix Monitor Tests Github Action (#2135) Valeriu Predoi
- [condalock] update conda lock file (#2141) Valeriu Predoi
- [Condalock] make sure mamba/conda are at latest version by forcing a pinned mamba install (#2136) Valeriu Predoi
- Update code coverage orbs (#2206) Bouwe Andela
- Revisit the comment-triggered Github Actions test (#2243) Valeriu Predoi
- Remove workflow that runs Github Actions tests from PR comment (#2244) Valeriu Predoi

39.11 Improvements

- Merge v2.9.x into main (#2128) Manuel Schlund
- Fix typo in citation file (#2182) Bouwe Andela
- Cleaned and extended function that extracts datetimes from paths (#2181) Manuel Schlund
- Add file encoding (and some read modes) at open file step (#2219) Valeriu Predoi
- Check type of argument passed to `read_cmor_tables()` (#2217) Valeriu Predoi
- Dynamic HTML output for monitoring (#2062) Brei Soliño
- Use PyPI's trusted publishers authentication (#2269) Valeriu Predoi

CHAPTER

FORTY

V2.9.0

40.1 Highlights

It is now possible to use the Dask distributed scheduler, which can significantly reduce the run-time of recipes. Configuration examples and advice are available in *our documentation*. More work on improving the computational performance is planned, so please share your experiences, good and bad, with this new feature in ESMValGroup/ESMValCore#1763.

This release includes

40.2 Backwards incompatible changes

- Remove deprecated configuration options (#2056) Bouwe Andela
  - The module `esmvalcore.experimental.config` has been removed. To upgrade, import the module from *`esmvalcore.config`* instead.
  - The module `esmvalcore._config` has been removed. To upgrade, use *`esmvalcore.config`* instead.
  - The methods `esmvalcore.config.Session.to_config_user` and `esmvalcore.config.Session.from_config_user` have been removed. To upgrade, use *`esmvalcore.config.Session`* to access the configuration values directly.

40.3 Bug fixes

- Respect `ignore_warnings` settings from the *project configuration* in *`esmvalcore.dataset.Dataset.load()`* (#2046) Manuel Schlund
- Fixed usage of custom location for *custom CMOR tables* (#2052) Manuel Schlund
- Fix issue with writing index.html when *running a recipe* with `--resume-from` (#2055) Bouwe Andela
- Fixed bug in ICON CMORizer that lead to shifted time coordinates (#2038) Manuel Schlund
- Include - in allowed characters for bibtex references (#2097) Alistair Sellar
- Do not raise an exception if the requested version of a file is not available for all matching files on ESGF (#2105) Bouwe Andela

40.4 Computational performance improvements

- Add support for *configuring Dask distributed* (#2049, #2122) Bouwe Andela
- Make `esmvalcore.preprocessor.extract_levels()` lazy (#1761) Bouwe Andela
- Lazy implementation of `esmvalcore.preprocessor.multi_model_statistics()` and `esmvalcore.preprocessor.ensemble_statistics()` (#968 and #2087) Peter Kalverla
- Avoid realizing data in preprocessor function `esmvalcore.preprocessor.concatenate()` when cubes overlap (#2109) Bouwe Andela

40.5 Documentation

- Remove unneeded sphinxcontrib extension (#2047) Valeriu Predoi
- Show ESMValTool logo on PyPI webpage (#2065) Valeriu Predoi
- Fix gitter badge in README (#2118) Rémi Kazeroni
- Add changelog for v2.9.0 (#2088 and #2123) Bouwe Andela

40.6 Fixes for datasets

- Pass the `esmvalcore.config.Session` to fixes (#1988) Manuel Schlund
- ICON: Allowed specifying vertical grid information in recipe (#2067) Manuel Schlund
- Allow specifying `raw_units` for CESM2, EMAC, and ICON CMORizers (#2043) Manuel Schlund
- ICON: allow specifying horizontal grid file in recipe/extra facets (#2078) Manuel Schlund
- Fix tas/tos CMIP6: FIO, KACE, MIROC, IITM (#2061) Pep Cos
- Add fix for EC-Earth3-Veg tos calendar (#2100) Bouwe Andela
- Correct GISS-E2-1-G `tos` units (#2099) Bouwe Andela

40.7 Installation

- Drop support for Python 3.8 (#2053) Bouwe Andela
- Add python 3.11 to Github Actions package (conda and PyPI) installation tests (#2083) Valeriu Predoi
- Remove `with_mypy` or `with-mypy` optional tool for prospector (#2108) Valeriu Predoi

40.8 Preprocessor

- Added `period='hourly'` for *esmvalcore.preprocessor.climate_statistics()* and *esmvalcore.preprocessor.anomalies()* (#2068) Manuel Schlund
- Support IPCC AR6 regions in *esmvalcore.preprocessor.extract_shape()* (#2008) Manuel Schlund

CHAPTER

FORTYONE

V2.8.1

41.1 Highlights

This release adds support for Python 3.11 and includes several bugfixes.

This release includes:

41.2 Bug fixes

- Pin numpy !=1.24.3 (#2011) Valeriu Predoi
- Fix a bug in recording provenance for the `mask_multimodel` preprocessor (#1984) Manuel Schlund
- Fix ICON hourly data rounding issues (#2022) Julian Bauer
- Use the default SSL context when using the `extract_location` preprocessor (#2023) Emma Hogan
- Make time-related CMOR fixes work with time dimensions *time1*, *time2*, *time3* (#1971) Manuel Schlund
- Always create a cache directory for storing ICON grid files (#2030) Manuel Schlund
- Fixed altitude <–> pressure level conversion for masked arrays in the `extract_levels` preprocessor (#1999) Manuel Schlund
- Allowed ignoring of scalar time coordinates in the `multi_model_statistics` preprocessor (#1961) Manuel Schlund

41.3 Fixes for datasets

- Add support for hourly ICON data (#1990) Julian Bauer
- Fix areacello in BCC-CSM2-MR (#1993) Rémi Kazeroni

41.4 Installation

- Add support for Python=3.11 (#1832) Valeriu Predoi
- Modernize conda lock file creation workflow with mamba, Mambaforge etc (#2027) Valeriu Predoi
- Pin *libnetcdf!=4.9.1* (#2072) Rémi Kazeroni

41.5 Documentation

- Add changelog for v2.8.1 (#2079) Bouwe Andela

41.6 Automatic testing

- Use mocked *geopy.geocoders.Nominatim* to avoid *ReadTimeoutError* (#2005) Manuel Schlund
- Update pre-commit hooks (#2020) Bouwe Andela

CHAPTER

FORTYTWO

V2.8.0

42.1 Highlights

- ESMValCore now supports wildcards in recipes and offers improved support for ancillary variables and dataset versioning thanks to contributions by Bouwe Andela. For details, see *Automatically populating a recipe with all available datasets* and *Defining supplementary variables*.
- Support for CORDEX datasets in a rotated pole coordinate system has been added by sloosvel.
- Native *ICON* output is now made UGRID-compliant on-the-fly to unlock the use of more sophisticated regridding algorithms, thanks to Manuel Schlund.
- The Python API has been extended with the addition of three modules: `esmvalcore.config`, `esmvalcore.dataset`, and `esmvalcore.local`, all these features courtesy of Bouwe Andela. For details, see our new example *Example notebooks*.
- The preprocessor `multi_model_statistics()` has been extended to support more use-cases thanks to contributions by Manuel Schlund. For details, see *Multi-model statistics*.

This release includes:

42.2 Backwards incompatible changes

Please read the descriptions of the linked pull requests for detailed upgrade instructions.

- The algorithm for automatically defining the ancillary variables and cell measures has been improved (#1609) Bouwe Andela. If this does not work as expected, more examples of how to adapt your recipes are given here and in the corresponding sections of the *recipe documentation* and the *preprocessor documentation*.
- Remove deprecated features scheduled for removal in v2.8.0 or earlier (#1826) Manuel Schlund. Removed `esmvalcore.iris_helpers.var_name_constraint` (has been deprecated in v2.6.0; please use `iris.NameConstraint` with the keyword argument `var_name` instead) and the option `always_use_ne_mask` for `esmvalcore.preprocessor.mask_landsea()` (has been deprecated in v2.5.0; the same behavior can now be achieved by specifying `supplementary_variables`.
- No files will be found if a non-existent version of a dataset is specified (#1835) Bouwe Andela. If a `version` of a dataset is specified in the recipe, the tool will now search for exactly that version, instead of simply using the latest version. Therefore, it is necessary to make sure that the version number in the directory tree matches with the version number in the recipe to find the files.
- The default filename template for obs4MIPs has been updated to better match filenames used in this project in (#1866) Bouwe Andela. This may cause issues if you are storing all the files for obs4MIPs in a directory with no subdirectories per dataset.

42.3 Deprecations

Please read the descriptions of the linked pull requests for detailed upgrade instructions.

- Various configuration related options that are now available through *esmvalcore.config* have been deprecated (#1769) Bouwe Andela.
- The `fx_variables` preprocessor argument and related features have been deprecated (#1609) Bouwe Andela. See here for more information.
- Combined `offline` and `always_search_esgf` into a single option `search_esgf` (#1935) Manuel Schlund. The configuration option/command line argument `offline` has been deprecated in favor of `search_esgf`. The previous `offline: true` is now `search_esgf: never` (the default); the previous `offline: false` is now `search_esgf: when_missing`. More details on how to adapt your workflow regarding these new options are given in #1935 and the documentation.
- `esmvalcore.preprocessor.cleanup()` has been deprecated (#1949) Manuel Schlund. Please do not use this anymore in the recipe (it is not necessary).

42.4 Python API

- Support searching ESGF for a specific version of a file and add *esmvalcore.esgf.ESGFFile.facets* (#1822) Bouwe Andela
- Fix issues with searching for files on ESGF (#1863) Bouwe Andela
- Move the `esmvalcore.experimental.config` module to *esmvalcore.config* (#1769) Bouwe Andela
- Add *esmvalcore.local*, a module to search data on the local filesystem (#1835) Bouwe Andela
- Add *esmvalcore.dataset* module (#1877) Bouwe Andela

42.5 Bug fixes

- Import from *esmvalcore.config* in the `esmvalcore.experimental` module (#1816) Bouwe Andela
- Added scalar coords of input cubes to output of esmpy_regrid (#1811) Manuel Schlund
- Fix severe bug in *esmvalcore.preprocessor.mask_fillvalues()* (#1823) Manuel Schlund
- Fix LWP of ICON on-the-fly CMORizer (#1839) Manuel Schlund
- Fixed issue in irregular regridding regarding scalar coordinates (#1845) Manuel Schlund
- Update product attributes and *metadata.yml* with cube metadata before saving files (#1837) Manuel Schlund
- Remove an extra space character from a filename (#1883) Bouwe Andela
- Improve resilience of ESGF search (#1869) Bouwe Andela
- Fix issue with no files found if timerange start/end differs in length (#1880) Bouwe Andela
- Add *driver* and *sub_experiment* tags to generate dataset aliases (#1886) sloosvel
- Fixed time points of native CESM2 output (#1772) Manuel Schlund
- Fix type hints for Python versions < 3.10 (#1897) Bouwe Andela
- Fixed *set_range_in_0_360* for dask arrays (#1919) Manuel Schlund

- Made equalized attributes in concatenated cubes consistent across runs (#1783) Manuel Schlund
- Fix issue with reading dates from files (#1936) Bouwe Andela
- Add institute name used on ESGF for CMIP5 CanAM4, CanCM4, and CanESM2 (#1937) Bouwe Andela
- Fix issue where data was not loaded and saved (#1962) Bouwe Andela
- Fix type hints for Python 3.8 (#1795) Bouwe Andela
- Update the institute facet of the CSIRO-Mk3L-1-2 model (#1966) Rémi Kazeroni
- Fixed race condition that may result in errors in `esmvalcore.preprocessor.cleanup()` (#1949) Manuel Schlund
- Update notebook so it uses supplementaries instead of ancillaries (#1945) Bouwe Andela

42.6 Documentation

- Fix anaconda badge in README (#1759) Valeriu Predoi
- Fix mistake in the documentation of *esmvalcore.esgf.find_files* (#1784) Bouwe Andela
- Support linking to "stable" ESMValTool version on readthedocs (#1608) Bouwe Andela
- Updated ICON doc with information on usage of extract_levels preprocessor (#1903) Manuel Schlund
- Add changelog for latest released version v2.7.1 (#1905) Valeriu Predoi
- Update *preprocessor.rst* due to renaming of NCEP dataset to NCEP-NCAR-R1 (#1908) Birgit Hassler
- Replace timerange nested lists in docs with overview table (#1940) Klaus Zimmermann
- Updated section "backward compatibility" in *contributing.rst* (#1918) Axel Lauer
- Add link to ESMValTool release procedure steps (#1957) Rémi Kazeroni
- Synchronize documentation table of contents with ESMValTool (#1958) Bouwe Andela

42.7 Improvements

- Support wildcards in the recipe and improve support for ancillary variables and dataset versioning (#1609) Bouwe Andela. More details on how to adapt your recipes are given in the corresponding pull request description and in the corresponding sections of the recipe documentation and the preprocessor documentation.
- Create a session directory with suffix "-1", "-2", etc if it already exists (#1818) Bouwe Andela
- Message for users when they use esmvaltool executable from esmvalcore only (#1831) Valeriu Predoi
- Order recipe output in index.html (#1899) Bouwe Andela
- Improve reading facets from ESGF search results (#1920) Bouwe Andela

42.8 Fixes for datasets

- Fix rotated coordinate grids and *tas* and *pr* for CORDEX datasets (#1765) sloosvel
- Made ICON output UGRID-compliant (on-the-fly) (#1664) Manuel Schlund
- Fix automatic download of ICON grid file and make ICON UGRIDization optional (*default: true*) (#1922) Manuel Schlund
- Add siconc fixes for EC-Earth3-Veg and EC-Earth3-Veg-LR models (#1771) Evgenia Galytska
- Fix siconc in KIOST-ESM (#1829) Lisa Bock
- Extension of ERA5 CMORizer (variable cl) (#1850) Axel Lauer
- Add standard variable names for EMAC (#1853) FranziskaWinterstein
- Fix for FGOALS-f3-L clt (#1928) Lisa Bock

42.9 Installation

- Add all deps to the conda-forge environment and suppress installing and reinstalling deps with pip at readthedocs builds (#1786) Valeriu Predoi
- Pin netCDF4<1.6.1 (#1805) Bouwe Andela
- Unpin NetCF4 (#1814) Valeriu Predoi
- Unpin flake8 (#1820) Valeriu Predoi
- Add iris-esmf-regrid as a dependency (#1809) sloosvel
- Pin esmpy<8.4 (#1871) Klaus Zimmermann
- Update esmpy import for ESMF v8.4.0 (#1876) Bouwe Andela

42.10 Preprocessor

- Allow *esmvalcore.preprocessor.multi_model_statistics()* on cubes with arbitrary dimensions (#1808) Manuel Schlund
- Smarter removal of coordinate metadata in *esmvalcore.preprocessor.multi_model_statistics()* preprocessor (#1813) Manuel Schlund
- Allowed usage of *esmvalcore.preprocessor.multi_model_statistics()* on single cubes/products (#1849) Manuel Schlund
- Allowed usage of *esmvalcore.preprocessor.multi_model_statistics()* on cubes with identical `name()` and `units` (but e.g. different long_name) (#1921) Manuel Schlund
- Allowed ignoring scalar coordinates in *esmvalcore.preprocessor.multi_model_statistics()* (#1934) Manuel Schlund
- Refactored *esmvalcore.preprocessor.regrid()* and removed unnecessary code not needed anymore due to new iris version (#1898) Manuel Schlund
- Do not realise coordinates during CMOR check (#1912) sloosvel
- Make *esmvalcore.preprocessor.extract_volume()* work with closed and mixed intervals and allow nearest value selection (#1930) sloosvel

42.11 Release

- Changelog for *v2.8.0rc1* (#1952) Rémi Kazeroni
- Increase version number for ESMValCore *v2.8.0rc1* (#1955) Rémi Kazeroni
- Changelog for *v2.8.0rc2* (#1959) Rémi Kazeroni
- Increase version number for ESMValCore *v2.8.0rc2* (#1973) Rémi Kazeroni
- Changelog for *v2.8.0* (#1978) Rémi Kazeroni
- Increase version number for ESMValCore *v2.8.0* (#1983) Rémi Kazeroni

42.12 Automatic testing

- Set implicit optional to true in *mypy* config to avert side effects and test fails from new mypy version (#1790) Valeriu Predoi
- Remove duplicate *implicit_optional = True* line in `setup.cfg` (#1791) Valeriu Predoi
- Fix failing test due to missing sample data (#1797) Bouwe Andela
- Remove outdated cmor_table facet from data finder tests (#1798) Bouwe Andela
- Modernize tests for *esmvalcore.preprocessor.save()* (#1799) Bouwe Andela
- No more sequential tests since SegFaults were not noticed anymore (#1819) Valeriu Predoi
- Update pre-commit configuration (#1821) Bouwe Andela
- Updated URL of ICON grid file used for testing (#1914) Manuel Schlund

42.13 Variable Derivation

- Add derivation of sea ice extent (#1695) sloosvel

CHAPTER

FORTYTHREE

V2.7.1

43.1 Highlights

This is a bugfix release where we unpin *cf-units* to allow the latest *iris=3.4.0* to be installed. It also includes an update to the default configuration used when searching the ESGF for files, to account for a recent change of the CEDA ESGF index node hostname. The changelog contains only changes that were made to the `main` branch.

43.2 Installation

- Set the version number on the development branches to one minor version more than the previous release (#1854) Bouwe Andela
- Unpin cf-units (#1770) Bouwe Andela

43.3 Bug fixes

- Improve error handling if an esgf index node is offline (#1834) Bouwe Andela

43.4 Automatic testing

- Removed unnecessary test that fails with iris 3.4.0 (#1846) Manuel Schlund
- Update CEDA ESGF index node hostname (#1838) Valeriu Predoi

CHAPTER

FORTYFOUR

V2.7.0

44.1 Highlights

- We have a new preprocessor function called 'rolling_window_statistics' implemented by Liza Malinina
- We have improved the support for native models, refactored native model fixes by adding common base class *NativeDatasetFix*, changed default DRS for reading native ICON output, and added tests for input/output filenames for ICON and EMAC on-the-fly CMORizer, all these features courtesy of Manuel Schlund
- Performance of preprocessor functions that use time dimensions has been sped up by **two orders of magnitude** thanks to contributions by Bouwe Andela

This release includes:

44.2 Backwards incompatible changes

- Change default DRS for reading native ICON output (#1705) Manuel Schlund

44.3 Bug fixes

- Add support for regions stored as MultiPolygon to extract_shape preprocessor (#1670) Bouwe Andela
- Fixed type annotations for Python 3.8 (#1700) Manuel Schlund
- Core *_io.concatenate()* may fail due to case when one of the cubes is scalar - this fixes that (#1715) Valeriu Predoi
- Pick up esmvalcore badge instead of esmvaltool one in README (#1749) Valeriu Predoi
- Restore support for scalar cubes to time selection preprocessor functions (#1750) Bouwe Andela
- Fix calculation of precipitation flux in EMAC on-the-fly CMORizer (#1755) Manuel Schlund

44.4 Deprecations

- Remove deprecation warning for regrid schemes already deprecated for v2.7.0 (#1753) Valeriu Predoi

44.5 Documentation

- Add Met Office Installation Method (#1692) mo-tgeddes
- Add MO-paths to config file (#1709) mo-tgeddes
- Update MO obs4MIPs paths in the user configuration file (#1734) mo-tgeddes
- Update *Making a release* section of the documentation (#1689) sloosvel
- Added changelog for v2.7.0 (#1746) Valeriu Predoi
- update CITATION.cff file with 2.7.0 release info (#1757) Valeriu Predoi

44.6 Improvements

- New preprocessor function 'rolling_window_statistics' (#1702) Liza Malinina
- Remove *pytest_flake8* plugin and use *flake8* instead (#1722) Valeriu Predoi
- Added CESM2 CMORizer (#1678) Manuel Schlund
- Speed up functions that use time dimension (#1713) Bouwe Andela
- Modernize and minimize pylint configuration (#1726) Bouwe Andela

44.7 Fixes for datasets

- Refactored native model fixes by adding common base class *NativeDatasetFix* (#1694) Manuel Schlund

44.8 Installation

- Pin *netCDF4 != 1.6.1* since that seems to throw a flurry of Segmentation Faults (#1724) Valeriu Predoi

44.9 Automatic testing

- Pin *flake8<5.0.0* since Circle CI tests are failing copiously (#1698) Valeriu Predoi
- Added tests for input/output filenames for ICON and EMAC on-the-fly CMORizer (#1718) Manuel Schlund
- Fix failed tests for Python<3.10 resulting from typing (#1748) Manuel Schlund

CHAPTER

FORTYFIVE

V2.6.0

45.1 Highlights

- A new set of CMOR fixes is now available in order to load native EMAC model output and CMORize it on the fly. For details, see *Supported native models: EMAC*.
- The version number of ESMValCore is now automatically generated using setuptools_scm, which extracts Python package versions from git metadata.

This release includes

45.2 Deprecations

- Deprecate the function *esmvalcore.var_name_constraint* (#1592) Manuel Schlund. This function is scheduled for removal in v2.8.0. Please use `iris.NameConstraint` with the keyword argument *var_name* instead: this is an exact replacement.

45.3 Bug fixes

- Added *start_year* and *end_year* attributes to derived variables (#1547) Manuel Schlund
- Show all results on recipe results webpage (#1560) Bouwe Andela
- Regridding regular grids with similar coordinates (#1567) Tomas Lovato
- Fix timerange wildcard search when deriving variables or downloading files (#1562) sloosvel
- Fix *force_derivation* bug (#1627) sloosvel
- Correct *build-and-deploy-on-pypi* action (#1634) sloosvel
- Apply *clip_timerange* to time dependent fx variables (#1603) sloosvel
- Correctly handle requests.exceptions.ConnectTimeout when an ESGF index node is offline (#1638) Bouwe Andela

45.4 CMOR standard

- Added custom CMOR tables used for EMAC CMORizer (#1599) Manuel Schlund
- Extended ICON CMORizer (#1549) Manuel Schlund
- Add CMOR check exception for a basin coord named sector (#1612) David Hohn
- Custom user-defined location for custom CMOR tables (#1625) Manuel Schlund

45.5 Containerization

- Remove update command in Dockerfile (#1630) sloosvel

45.6 Community

- Add David Hohn to contributors' list (#1586) Valeriu Predoi

45.7 Documentation

- [Github Actions Docs] Full explanation on how to use the GA test triggered by PR comment and added docs link for GA hosted runners (#1553) Valeriu Predoi
- Update the command for building the documentation (#1556) Bouwe Andela
- Update documentation on running the tool (#1400) Bouwe Andela
- Add support for DKRZ-Levante (#1558) Rémi Kazeroni
- Improved documentation on native dataset support (#1559) Manuel Schlund
- Tweak *extract_point* preprocessor: explain what it returns if one point coord outside cube and add explicit test (#1584) Valeriu Predoi
- Update CircleCI, readthedocs, and Docker configuration (#1588) Bouwe Andela
- Remove support for Mistral in *config-user.yml* (#1620) Rémi Kazeroni
- Add changelog for v2.6.0rc1 (#1633) sloosvel
- Add a note on transferring permissions to the release manager (#1645) Bouwe Andela
- Add documentation on building and uploading Docker images (#1644) Bouwe Andela
- Update documentation on ESMValTool module at DKRZ (#1647) Rémi Kazeroni
- Expanded information on deprecations in changelog (#1658) Manuel Schlund

45.8 Improvements

- Removed trailing whitespace in custom CMOR tables (#1564) Manuel Schlund
- Try searching multiple ESGF index nodes (#1561) Bouwe Andela
- Add CMIP6 *amoc* derivation case and add a test (#1577) Valeriu Predoi
- Added EMAC CMORizer (#1554) Manuel Schlund
- Improve performance of *volume_statistics* (#1545) sloosvel

45.9 Fixes for datasets

- Fixes of ocean variables in multiple CMIP6 datasets (#1566) Tomas Lovato
- Ensure lat/lon bounds in FGOALS-l3 atmos variables are contiguous (#1571) sloosvel
- Added *AllVars* fix for CMIP6's ICON-ESM-LR (#1582) Manuel Schlund

45.10 Installation

- Removed *package/meta.yml* (#1540) Manuel Schlund
- Pinned iris>=3.2.1 (#1552) Manuel Schlund
- Use setuptools-scm to automatically generate the version number (#1578) Bouwe Andela
- Pin cf-units to lower than 3.1.0 to temporarily avoid changes within new version related to calendars (#1659) Valeriu Predoi

45.11 Preprocessor

- Allowed special case for unit conversion of precipitation (*kg m-2 s-1 <–> mm day-1*) (#1574) Manuel Schlund
- Add general *extract_coordinate_points* preprocessor (#1581) sloosvel
- Add preprocessor *accumulate_coordinate* (#1281) Javier Vegas-Regidor
- Add *axis_statistics* and improve *depth_integration* (#1589) sloosvel

45.12 Release

- Increase version number for ESMValCore v2.6.0rc1 (#1632) sloosvel
- Update changelog and version for 2.6rc3 (#1646) sloosvel
- Add changelog for rc4 (#1662) sloosvel

45.13 Automatic testing

- Refresh CircleCI cache weekly (#1597) Bouwe Andela
- Use correct cache restore key on CircleCI (#1598) Bouwe Andela
- Install git and ssh before checking out code on CircleCI (#1601) Bouwe Andela
- Fetch all history in Github Action tests (#1622) sloosvel
- Test Github Actions dashboard badge from meercode.io (#1640) Valeriu Predoi
- Improve esmvalcore.esgf unit test (#1650) Bouwe Andela

45.14 Variable Derivation

- Added derivation of *hfns* (#1594) Manuel Schlund

CHAPTER

FORTYSIX

V2.5.0

46.1 Highlights

- The new preprocessor `extract_location()` can extract arbitrary locations on the Earth using the geopy package that connects to OpenStreetMap. For details, see *Extract location*.
- Time ranges can now be extracted using the ISO 8601 format. In addition, wildcards are allowed, which makes the time selection much more flexible. For details, see *Recipe section: Datasets*.
- The new preprocessor `ensemble_statistics()` can calculate arbitrary statistics over all ensemble members of a simulation. In addition, the preprocessor `multi_model_statistics()` now accepts the keyword `groupy`, which allows the calculation of multi-model statistics over arbitrary multi-model ensembles. For details, see *Ensemble statistics* and *Multi-model statistics*.

This release includes

46.2 Backwards incompatible changes

- Update Cordex section in *config-developer.yml* (#1303) francesco-cmcc. This changes the naming convention of ESMValCore's output files from CORDEX dataset. This only affects recipes that use CORDEX data. Most likely, no changes in diagnostics are necessary; however, if code relies on the specific naming convention of files, it might need to be adapted.
- Dropped Python 3.7 (#1530) Manuel Schlund. ESMValCore v2.5.0 dropped support for Python 3.7. From now on Python >=3.8 is required to install ESMValCore. The main reason for this is that conda-forge dropped support for Python 3.7 for OSX and arm64 (more details are given here).

46.3 Bug fixes

- Fix *extract_shape* when fx vars are present (#1403) sloosvel
- Added support of *extra_facets* to fx variables added by the preprocessor (#1399) Manuel Schlund
- Augmented input for derived variables with extra_facets (#1412) Manuel Schlund
- Correctly use masked arrays after *unstructured_nearest* regridding (#1414) Manuel Schlund
- Fixing the broken derivation script for XCH4 (and XCO2) (#1428) Birgit Hassler
- Ignore *.pymon-journal* file in test discovery (#1436) Valeriu Predoi
- Fixed bug that caused automatic download to fail in rare cases (#1442) Manuel Schlund

- Add new *JULIA_LOAD_PATH* to diagnostic task test (#1444) Valeriu Predoi
- Fix provenance file permissions (#1468) Bouwe Andela
- Fixed usage of *statistics=std_dev* option in multi-model statistics preprocessors (#1478) Manuel Schlund
- Removed scalar coordinates *p0* and *ptop* prior to merge in *multi_model_statistics* (#1471) Axel Lauer
- Added *dataset* and *alias* attributes to *multi_model_statistics* output (#1483) Manuel Schlund
- Fixed issues with multi-model-statistics timeranges (#1486) Manuel Schlund
- Fixed output messages for CMOR logging (#1494) Manuel Schlund
- Fixed *clip_timerange* if only a single time point is extracted (#1497) Manuel Schlund
- Fixed chunking in *multi_model_statistics* (#1500) Manuel Schlund
- Fixed renaming of auxiliary coordinates in *multi_model_statistics* if coordinates are equal (#1502) Manuel Schlund
- Fixed timerange selection for automatic downloads (#1517) Manuel Schlund
- Fixed chunking in *multi_model_statistics* (#1524) Manuel Schlund

46.4 Deprecations

- Renamed vertical regridding schemes (#1429) Manuel Schlund. Old regridding schemes are supported until v2.7.0. For details, see *Vertical interpolation schemes*.

46.5 Documentation

- Remove duplicate entries in changelog (#1391) Klaus Zimmermann
- Documentation on how to use HPC central installations (#1409) Valeriu Predoi
- Correct brackets in preprocessor documentation for list of seasons (#1420) Bouwe Andela
- Add Python=3.10 to package info, update Circle CI auto install and documentation for Python=3.10 (#1432) Valeriu Predoi
- Reverted unintentional change in *.zenodo.json* (#1452) Manuel Schlund
- Synchronized config-user.yml with version from ESMValTool (#1453) Manuel Schlund
- Solved issues in configuration files (#1457) Manuel Schlund
- Add direct link to download conda lock file in the install documentation (#1462) Valeriu Predoi
- CITATION.cff fix and automatic validation of citation metadata (#1467) Valeriu Predoi
- Updated documentation on how to deprecate features (#1426) Manuel Schlund
- Added reference hook to conda lock in documentation install section (#1473) Valeriu Predoi
- Increased ESMValCore version to 2.5.0rc1 (#1477) Manuel Schlund
- Added changelog for v2.5.0 release (#1476) Manuel Schlund
- Increased ESMValCore version to 2.5.0rc2 (#1487) Manuel Schlund
- Added some authors to citation and zenodo files (#1488) SarahAlidoost
- Restored *scipy* intersphinx mapping (#1491) Manuel Schlund

- Increased ESMValCore version to 2.5.0rc3 (#1504) Manuel Schlund
- Fix download instructions for the MSWEP dataset (#1506) Rémi Kazeroni
- Documentation updated for the new cmorizer framework (#1417) Rémi Kazeroni
- Added tests for duplicates in changelog and removed duplicates (#1508) Manuel Schlund
- Increased ESMValCore version to 2.5.0rc4 (#1519) Manuel Schlund
- Add Github Actions Test badge in README (#1526) Valeriu Predoi
- Increased ESMValCore version to 2.5.0rc5 (#1529) Manuel Schlund
- Increased ESMValCore version to 2.5.0rc6 (#1532) Manuel Schlund

46.6 Fixes for datasets

- Added fix for AIRS v2.1 (obs4mips) (#1472) Axel Lauer

46.7 Preprocessor

- Added bias preprocessor (#1406) Manuel Schlund
- Improve error messages when a preprocessor is failing (#1408) Manuel Schlund
- Added option to explicitly not use fx variables in preprocessors (#1416) Manuel Schlund
- Add *extract_location* preprocessor to extract town, city, mountains etc - anything specifiable by a location (#1251) Javier Vegas-Regidor
- Add ensemble statistics preprocessor and 'groupby' option for multimodel (#673) sloosvel
- Generic regridding preprocessor (#1448) Klaus Zimmermann

46.8 Automatic testing

- Add *pandas* as dependency :panda_face: (#1402) Valeriu Predoi
- Fixed tests for python 3.7 (#1410) Manuel Schlund
- Remove accessing *.xml()* cube method from test (#1419) Valeriu Predoi
- Remove flag to use pip 2020 solver from Github Action pip install command on OSX (#1357) Valeriu Predoi
- Add Python=3.10 to Github Actions and switch to Python=3.10 for the Github Action that builds the PyPi package (#1430) Valeriu Predoi
- Pin *flake8<4* to keep getting relevant error traces when tests fail with FLAKE8 issues (#1434) Valeriu Predoi
- Implementing conda lock (#1164) Valeriu Predoi
- Relocate *pytest-monitor* outputted database *.pymon* so *.pymon-journal* file should not be looked for by *pytest* (#1441) Valeriu Predoi
- Switch to Mambaforge in Github Actions tests (#1438) Valeriu Predoi
- Turn off conda lock file creation on any push on *main* branch from Github Action test (#1489) Valeriu Predoi
- Add DRS path test for IPSLCM files (#1490) Stéphane Sénési

- Add a test module that runs tests of *iris* I/O every time we notice serious bugs there (#1510) Valeriu Predoi
- [Github Actions] Trigger Github Actions tests (*run-tests.yml* workflow) from a comment in a PR (#1520) Valeriu Predoi
- Update Linux condalock file (various pull requests) github-actions[bot]

46.9 Installation

- Move *nested-lookup* dependency to *environment.yml* to be installed from conda-forge instead of PyPi (#1481) Valeriu Predoi
- Pinned *iris* (#1511) Manuel Schlund
- Updated dependencies (#1521) Manuel Schlund
- Pinned iris<3.2.0 (#1525) Manuel Schlund

46.10 Improvements

- Allow to load all files, first X years or last X years in an experiment (#1133) sloosvel
- Filter tasks earlier (#1264) Javier Vegas-Regidor
- Added earlier validation for command line arguments (#1435) Manuel Schlund
- Remove *profile_diagnostic* from diagnostic settings and increase test coverage of *_task.py* (#1404) Valeriu Predoi
- Add *output2* to the *product* extra facet of CMIP5 data (#1514) Rémi Kazeroni
- Speed up ESGF search (#1512) Bouwe Andela

CHAPTER

FORTYSEVEN

V2.4.0

47.1 Highlights

- ESMValCore now has the ability to automatically download missing data from ESGF. For details, see *Data Retrieval*.
- ESMValCore now also can resume an earlier run. This is useful to re-use expensive preprocessor results. For details, see *Running*.

This release includes

47.2 Bug fixes

- Crop on the ID-selected region(s) and not on the whole shapefile (#1151) Stef Smeets
- Add 'comment' to list of removed attributes (#1244) Peter Kalverla
- Speed up multimodel statistics and fix bug in peak computation (#1301) Bouwe Andela
- No longer make plots of provenance (#1307) Bouwe Andela
- No longer embed provenance in output files (#1306) Bouwe Andela
- Removed automatic addition of areacello to obs4mips datasets (#1316) Manuel Schlund
- Pin docutils <0.17 to fix bullet lists on readthedocs (#1320) Klaus Zimmermann
- Fix obs4MIPs capitalization (#1328) Bouwe Andela
- Fix Python 3.7 tests (#1330) Bouwe Andela
- Handle fx variables in *extract_levels* and some time operations (#1269) sloosvel
- Refactored mask regridding for irregular grids (fixes #772) (#865) Klaus Zimmermann
- Fix *da.broadcast_to* call when the fx cube has different shape than target data cube (#1350) Valeriu Predoi
- Add tests for _aggregate_time_fx (#1354) sloosvel
- Fix extra facets (#1360) Bouwe Andela
- Pin pip!=21.3 to avoid pypa/pip#10573 with editable installs (#1359) Klaus Zimmermann
- Add a custom *date2num* function to deal with changes in cftime (#1373) Klaus Zimmermann
- Removed custom version of *AtmosphereSigmaFactory* (#1382) Manuel Schlund

47.3 Deprecations

- Remove write_netcdf and write_plots from config-user.yml (#1300) Bouwe Andela

47.4 Documentation

- Add link to plot directory in index.html (#1256) Stef Smeets
- Work around issue with yapf not following PEP8 (#1277) Bouwe Andela
- Update the core development team (#1278) Bouwe Andela
- Update the documentation of the provenance interface (#1305) Bouwe Andela
- Update version number to first release candidate 2.4.0rc1 (#1363) Klaus Zimmermann
- Update to new ESMValTool logo (#1374) Klaus Zimmermann
- Update version number for third release candidate 2.4.0rc3 (#1384) Klaus Zimmermann
- Update changelog for 2.4.0rc3 (#1385) Klaus Zimmermann
- Update version number to final 2.4.0 release (#1389) Klaus Zimmermann
- Update changelog for 2.4.0 (#1366) Klaus Zimmermann

47.5 Fixes for datasets

- Add fix for differing latitude coordinate between historical and ssp585 in MPI-ESM1-2-HR r2i1p1f1 (#1292) Bouwe Andela
- Add fixes for time and latitude coordinate of EC-Earth3 r3i1p1f1 (#1290) Bouwe Andela
- Apply latitude fix to all CCSM4 variables (#1295) Bouwe Andela
- Fix lat and lon bounds for FGOALS-g3 mrsos (#1289) Thomas Crocker
- Add grid fix for tos in fgoals-f3-l (#1326) sloosvel
- Add fix for CIESM pr (#1344) Bouwe Andela
- Fix DRS for IPSLCM : split attribute 'freq' into : 'out' and 'freq' (#1304) Stéphane Sénési - work

47.6 CMOR standard

- Remove history attribute from coords (#1276) Javier Vegas-Regidor
- Increased flexibility of CMOR checks for datasets with generic alevel coordinates (#1032) Manuel Schlund
- Automatically fix small deviations in vertical levels (#1177) Bouwe Andela
- Adding standard names to the custom tables of the *rlns* and *rsns* variables (#1386) Rémi Kazeroni

47.7 Preprocessor

- Implemented fully lazy climate_statistics (#1194) Manuel Schlund
- Run the multimodel statistics preprocessor last (#1299) Bouwe Andela

47.8 Automatic testing

- Improving test coverage for _task.py (#514) Valeriu Predoi
- Upload coverage to codecov (#1190) Bouwe Andela
- Improve codecov status checks (#1195) Bouwe Andela
- Fix curl install in CircleCI (#1228) Javier Vegas-Regidor
- Drop support for Python 3.6 (#1200) Valeriu Predoi
- Allow more recent version of *scipy* (#1182) Manuel Schlund
- Speed up conda build *conda_build* Circle test by using *mamba* solver via *boa* (and use it for Github Actions test too) (#1243) Valeriu Predoi
- Fix numpy deprecation warnings (#1274) Bouwe Andela
- Unpin upper bound for iris (previously was at <3.0.4) (#1275) Valeriu Predoi
- Modernize *conda_install* test on Circle CI by installing from conda-forge with Python 3.9 and change install instructions in documentation (#1280) Valeriu Predoi
- Run a nightly Github Actions workflow to monitor tests memory per test (configurable for other metrics too) (#1284) Valeriu Predoi
- Speed up tests of tasks (#1302) Bouwe Andela
- Fix upper case to lower case variables and functions for flake compliance in *tests/unit/preprocessor/_regrid/test_extract_levels.py* (#1347) Valeriu Predoi
- Cleaned up a bit Github Actions workflows (#1345) Valeriu Predoi
- Update circleci jobs: renaming tests to more descriptive names and removing conda build test (#1351) Klaus Zimmermann
- Pin iris to latest *>=3.1.0* (#1341) Valeriu Predoi

47.9 Installation

- Pin esmpy to anything but 8.1.0 since that particular one changes the CPU affinity (#1310) Valeriu Predoi

47.10 Improvements

- Add a more friendly and useful message when using default config file (#1233) Valeriu Predoi
- Replace os.walk by glob.glob in data finder (only look for data in the specified locations) (#1261) Bouwe Andela
- Machine-specific directories for auxiliary data in the *config-user.yml* file (#1268) Rémi Kazeroni
- Add an option to download missing data from ESGF (#1217) Bouwe Andela
- Speed up provenance recording (#1327) Bouwe Andela
- Improve results web page (#1332) Bouwe Andela
- Move institutes from config-developer.yml to default extra facets config and add wildcard support for extra facets (#1259) Bouwe Andela
- Add support for re-using preprocessor output from previous runs (#1321) Bouwe Andela
- Log fewer messages to screen and hide stack trace for known recipe errors (#1296) Bouwe Andela
- Log ESMValCore and ESMValTool versions when running (#1263) Javier Vegas-Regidor
- Add "grid" as a tag to the output file template for CMIP6 (#1356) Klaus Zimmermann
- Implemented ICON project to read native ICON model output (#1079) Brei Soliño

CHAPTER

FORTYEIGHT

V2.3.1

This release includes

48.1 Bug fixes

- Update config-user.yml template with correct drs entries for CEDA-JASMIN (#1184) Valeriu Predoi
- Enhancing MIROC5 fix for hfls and evspsbl (#1192) katjaweigel
- Fix alignment of daily data with inconsistent calendars in multimodel statistics (#1212) Peter Kalverla
- Pin cf-units, remove github actions test for Python 3.6 and fix test_access1_0 and test_access1_3 to use cf-units for comparisons (#1197) Valeriu Predoi
- Fixed search for fx files when no `mip` is given (#1216) Manuel Schlund
- Make sure climate statistics always returns original dtype (#1237) Klaus Zimmermann
- Bugfix for regional regridding when non-integer range is passed (#1231) Stef Smeets
- Make sure area_statistics preprocessor always returns original dtype (#1239) Klaus Zimmermann
- Add "." (dot) as allowed separation character for the time range group (#1248) Klaus Zimmermann

48.2 Documentation

- Add a link to the instructions to use pre-installed versions on HPC clusters (#1186) Rémi Kazeroni
- Bugfix release: set version to 2.3.1 (#1253) Klaus Zimmermann

48.3 Fixes for datasets

- Set circular attribute in MCM-UA-1-0 fix (#1178) sloosvel
- Fixed time coordinate of MIROC-ESM (#1188) Manuel Schlund

48.4 Preprocessor

- Filter warnings about collapsing multi-model dimension in multimodel statistics preprocessor function (#1215) Bouwe Andela
- Remove fx variables before computing multimodel statistics (#1220) sloosvel

48.5 Installation

- Pin lower bound for iris to 3.0.2 (#1206) Valeriu Predoi
- Pin *iris<3.0.4* to ensure we still (sort of) support Python 3.6 (#1252) Valeriu Predoi

48.6 Improvements

- Add test to verify behaviour for scalar height coord for tas in multi-model (#1209) Peter Kalverla
- Sort missing years in "No input data available for years" message (#1225) Lee de Mora

CHAPTER

FORTYNINE

V2.3.0

This release includes

49.1 Bug fixes

- Extend preprocessor multi_model_statistics to handle data with “altitude” coordinate (#1010) Axel Lauer
- Remove scripts included with CMOR tables (#1011) Bouwe Andela
- Avoid side effects in extract_season (#1019) Javier Vegas-Regidor
- Use nearest scheme to avoid interpolation errors with masked data in regression test (#1021) Stef Smeets
- Move _get_time_bounds from preprocessor._time to cmor.check to avoid circular import with cmor module (#1037) Valeriu Predoi
- Fix test that makes conda build fail (#1046) Valeriu Predoi
- Fix ‘positive’ attribute for rsns/rlns variables (#1051) Lukas Brunner
- Added preprocessor mask_multimodel (#767) Manuel Schlund
- Fix bug when fixing bounds after fixing longitude values (#1057) sloosvel
- Run conda build parallel AND sequential tests (#1065) Valeriu Predoi
- Add key to id_prop (#1071) Lukas Brunner
- Fix bounds after reversing coordinate values (#1061) sloosvel
- Fixed –skip-nonexistent option (#1093) Manuel Schlund
- Do not consider CMIP5 variable sit to be the same as sithick from CMIP6 (#1033) Bouwe Andela
- Improve finding date range in filenames (enforces separators) (#1145) Stéphane Sénési - work
- Review fx handling (#1147) sloosvel
- Fix lru cache decorator with explicit call to method (#1172) Valeriu Predoi
- Update _volume.py (#1174) Lee de Mora

49.2 Deprecations

49.3 Documentation

- Final changelog for 2.3.0 (#1163) Klaus Zimmermann
- Set version to 2.3.0 (#1162) Klaus Zimmermann
- Fix documentation build (#1006) Bouwe Andela
- Add labels required for linking from ESMValTool docs (#1038) Bouwe Andela
- Update contribution guidelines (#1047) Bouwe Andela
- Fix basestring references in documentation (#1106) Javier Vegas-Regidor
- Updated references master to main (#1132) Axel Lauer
- Add instructions how to use the central installation at DKRZ-Mistral (#1155) Rémi Kazeroni

49.4 Fixes for datasets

- Added fixes for various CMIP5 datasets, variable cl (3-dim cloud fraction) (#1017) Axel Lauer
- Added fixes for hybrid level coordinates of CESM2 models (#882) Manuel Schlund
- Extending LWP fix for CMIP6 models (#1049) Axel Lauer
- Add fixes for the net & up radiation variables from ERA5 (#1052) Lukas Brunner
- Add derived variable rsus (#1053) Lukas Brunner
- Supported *mip*-level fixes (#1095) Manuel Schlund
- Fix erroneous use of *grid_latitude* and *grid_longitude* and cleaned ocean grid fixes (#1092) Manuel Schlund
- Fix for pr of miroc5 (#1110) Rémi Kazeroni
- Ocean depth fix for cnrm_esm2_1, gfdl_esm4, ipsl_cm6a_lr datasets + mcm_ua_1_0 (#1098) Tomas Lovato
- Fix for uas variable of the MCM_UA_1_0 dataset (#1102) Rémi Kazeroni
- Fixes for sos and siconc of BCC models (#1090) Rémi Kazeroni
- Run fgco2 fix for all CESM2 models (#1108) Lisa Bock
- Fixes for the siconc variable of CMIP6 models (#1105) Rémi Kazeroni
- Fix wrong sign for land surface flux (#1113) Lisa Bock
- Fix for pr of EC_EARTH (#1116) Rémi Kazeroni

49.5 CMOR standard

- Format cmor related files (#976) Javier Vegas-Regidor
- Check presence of time bounds and guess them if needed (#849) sloosvel
- Add custom variable "tasaga" (#1118) Lisa Bock
- Find files for CMIP6 DCPP startdates (#771) sloosvel

49.6 Preprocessor

- Update tests for multimodel statistics preprocessor (#1023) Stef Smeets
- Raise in extract_season and extract_month if result is None (#1041) Javier Vegas-Regidor
- Allow selection of shapes in extract_shape (#764) Javier Vegas-Regidor
- Add option for regional regridding to regrid preprocessor (#1034) Stef Smeets
- Load fx variables as cube cell measures / ancillary variables (#999) sloosvel
- Check horizontal grid before regridding (#507) Benjamin Müller
- Clip irregular grids (#245) Bouwe Andela
- Use native iris functions in multi-model statistics (#1150) Peter Kalverla

49.7 Notebook API (experimental)

49.8 Automatic testing

- Report coverage for tests that run on any pull request (#994) Bouwe Andela
- Install ESMValTool sample data from PyPI (#998) Javier Vegas-Regidor
- Fix tests for multi-processing with spawn method (i.e. macOSX with Python>3.8) (#1003) Barbara Vreede
- Switch to running the Github Action test workflow every 3 hours in single thread mode to observe if Segmentation Faults occur (#1022) Valeriu Predoi
- Revert to original Github Actions test workflow removing the 3-hourly test run with -n 1 (#1025) Valeriu Predoi
- Avoid stale cache for multimodel statistics regression tests (#1030) Bouwe Andela
- Add newer Python versions in OSX to Github Actions (#1035) Barbara Vreede
- Add tests for type annotations with mypy (#1042) Stef Smeets
- Run problematic cmor tests sequentially to avoid segmentation faults on CircleCI (#1064) Valeriu Predoi
- Test installation of esmvalcore from conda-forge (#1075) Valeriu Predoi
- Added additional test cases for integration tests of data_finder.py (#1087) Manuel Schlund
- Pin cf-units and fix tests (cf-units>=2.1.5) (#1140) Valeriu Predoi
- Fix failing CircleCI tests (#1167) Bouwe Andela
- Fix test failing due to fx files chosen differently on different OS's (#1169) Valeriu Predoi

- Compare datetimes instead of strings in _fixes/cmip5/test_access1_X.py (#1173) Valeriu Predoi
- Pin Python to 3.9 in environment.yml on CircleCI and skip mypy tests in conda build (#1176) Bouwe Andela

49.9 Installation

- Update yamale to version 3 (#1059) Klaus Zimmermann

49.10 Improvements

- Refactor diagnostics / tags management (#939) Stef Smeets
- Support multiple paths in input_dir (#1000) Javier Vegas-Regidor
- Generate HTML report with recipe output (#991) Stef Smeets
- Add timeout to requests.get in _citation.py (#1091) SarahAlidoost
- Add SYNDA drs for CMIP5 and CMIP6 (closes #582) (#583) Klaus Zimmermann
- Add basic support for variable mappings (#1124) Klaus Zimmermann
- Handle IPSL-CM6 (#1153) Stéphane Sénési - work

CHAPTER

FIFTY

V2.2.0

50.1 Highlights

ESMValCore is now using the recently released Iris 3. We acknowledge that this change may impact your work, as Iris 3 introduces several changes that are not backward-compatible, but we think that moving forward is the best decision for the tool in the long term.

This release is also the first one including support for downloading CMIP6 data using Synda and we have also started supporting Python 3.9. Give it a try!

This release includes

50.2 Bug fixes

- Fix path settings for DKRZ/Mistral (#852) Bouwe Andela
- Change logic for calling the diagnostic script to avoid problems with scripts where the executable bit is accidentally set (#877) Bouwe Andela
- Fix overwriting in generic level check (#886) sloosvel
- Add double quotes to script args in rerun screen message when using vprof profiling (#897) Valeriu Predoi
- Simplify time handling in multi-model statistics preprocessor (#685) Peter Kalverla
- Fix links to Iris documentation (#966) Javier Vegas-Regidor
- Bugfix: Fix units for MSWEP data (#986) Stef Smeets

50.3 Deprecations

- Deprecate defining write_plots and write_netcdf in config-user file (#808) Bouwe Andela

50.4 Documentation

- Fix numbering of steps in release instructions (#838) Bouwe Andela
- Add labels to changelogs of individual versions for easy reference (#899) Klaus Zimmermann
- Make CircleCI badge specific to main branch (#902) Bouwe Andela
- Fix docker build badge url (#906) Stef Smeets
- Update github PR template (#909) Stef Smeets
- Refer to ESMValTool GitHub discussions page in the error message (#900) Bouwe Andela
- Support automatically closing issues (#922) Bouwe Andela
- Fix checkboxes in PR template (#931) Stef Smeets
- Change in config-user defaults and documentation with new location for esmeval OBS data on JASMIN (#958) Valeriu Predoi
- Update Core Team info (#942) Axel Lauer
- Update iris documentation URL for sphinx (#964) Bouwe Andela
- Set version to 2.2.0 (#977) Javier Vegas-Regidor
- Add first draft of v2.2.0 changelog (#983) Javier Vegas-Regidor
- Add checkbox in PR template to assign labels (#985) Javier Vegas-Regidor
- Update install.rst (#848) bascrezee
- Change the order of the publication steps (#984) Javier Vegas-Regidor
- Add instructions how to use esmvaltool from HPC central installations (#841) Valeriu Predoi

50.5 Fixes for datasets

- Fixing unit for derived variable rsnstcsnorm to prevent overcorrection2 (#846) katjaweigel
- Cmip6 fix awi cm 1 1 mr (#822) mwjury
- Cmip6 fix ec earth3 veg (#836) mwjury
- Changed latitude longitude fix from Tas to AllVars. (#916) katjaweigel
- Fix for precipitation (pr) to use ERA5-Land cmorizer (#879) katjaweigel
- Cmip6 fix ec earth3 (#837) mwjury
- Cmip6_fix_fgoals_f3_l_Amon_time_bnds (#831) mwjury
- Fix for FGOALS-f3-L sftlf (#667) mwjury
- Improve ACCESS-CM2 and ACCESS-ESM1-5 fixes and add CIESM and CESM2-WACCM-FV2 fixes for cl, clw and cli (#635) Axel Lauer
- Add fixes for cl, cli, clw and tas for several CMIP6 models (#955) Manuel Schlund
- Dataset fixes for MSWEP (#969) Stef Smeets
- Dataset fixes for: ACCESS-ESM1-5, CanESM5, CanESM5 for carbon cycle (#947) Bettina Gier
- Fixes for KIOST-ESM (CMIP6) (#904) Rémi Kazeroni

- Fixes for AWI-ESM-1-1-LR (CMIP6, piControl) (#911) Rémi Kazeroni

50.6 CMOR standard

- CMOR check generic level coordinates in CMIP6 (#598) sloosvel
- Update CMIP6 tables to 6.9.33 (#919) Javier Vegas-Regidor
- Adding custom variables for tas uncertainty (#924) Lisa Bock
- Remove monotonicity coordinate check for unstructured grids (#965) Javier Vegas-Regidor

50.7 Preprocessor

- Added clip_start_end_year preprocessor (#796) Manuel Schlund
- Add support for downloading CMIP6 data with Synda (#699) Bouwe Andela
- Add multimodel tests using real data (#856) Stef Smeets
- Add plev/altitude conversion to extract_levels (#892) Axel Lauer
- Add possibility of custom season extraction. (#247) mwjury
- Adding the ability to derive xch4 (#783) Birgit Hassler
- Add preprocessor function to resample time and compute x-hourly statistics (#696) Javier Vegas-Regidor
- Fix duplication in preprocessors DEFAULT_ORDER introduced in #696 (#973) Javier Vegas-Regidor
- Use consistent precision in multi-model statistics calculation and update reference data for tests (#941) Peter Kalverla
- Refactor multi-model statistics code to facilitate ensemble stats and lazy evaluation (#949) Peter Kalverla
- Add option to exclude input cubes in output of multimodel statistics to solve an issue introduced by #949 (#978) Peter Kalverla

50.8 Automatic testing

- Pin cftime>=1.3.0 to have newer string formatting in and fix two tests (#878) Valeriu Predoi
- Switched miniconda conda setup hooks for Github Actions workflows (#873) Valeriu Predoi
- Add test for latest version resolver (#874) Stef Smeets
- Update codacy coverage reporter to fix coverage (#905) Niels Drost
- Avoid hardcoded year in tests and add improvement to plev test case (#921) Bouwe Andela
- Pin scipy to less than 1.6.0 until ESMValGroup/ESMValCore/issues/927 gets resolved (#928) Valeriu Predoi
- Github Actions: change time when conda install test runs (#930) Valeriu Predoi
- Remove redundant test line from test_utils.py (#935) Valeriu Predoi
- Removed netCDF4 package from integration tests of fixes (#938) Manuel Schlund
- Use new conda environment for installing ESMValCore in Docker containers (#951) Bouwe Andela

50.9 Notebook API (experimental)

- Implement importable config object in experimental API submodule (#868) Stef Smeets
- Add loading and running recipes to the notebook API (#907) Stef Smeets
- Add displaying and loading of recipe output to the notebook API (#957) Stef Smeets
- Add functionality to run single diagnostic task to notebook API (#962) Stef Smeets

50.10 Improvements

- Create CODEOWNERS file (#809) Javier Vegas-Regidor
- Remove code needed for Python <3.6 (#844) Bouwe Andela
- Add requests as a dependency (#850) Bouwe Andela
- Pin Python to less than 3.9 (#870) Valeriu Predoi
- Remove numba dependency (#880) Manuel Schlund
- Add Listing and finding recipes to the experimental notebook API (#901) Stef Smeets
- Skip variables that don't have dataset or additional_dataset keys (#860) Valeriu Predoi
- Refactor logging configuration (#933) Stef Smeets
- Xco2 derivation (#913) Bettina Gier
- Working environment for Python 3.9 (pin to !=3.9.0) (#885) Valeriu Predoi
- Print source file when using config get_config_user command (#960) Valeriu Predoi
- Switch to Iris 3 (#819) Stef Smeets
- Refactor tasks (#959) Stef Smeets
- Restore task summary in debug log after #959 (#981) Bouwe Andela
- Pin pre-commit hooks (#974) Stef Smeets
- Improve error messages when data is missing (#917) Javier Vegas-Regidor
- Set remove_preproc_dir to false in default config-user (#979) Valeriu Predoi
- Move fiona to be installed from conda forge (#987) Valeriu Predoi
- Re-added fiona in setup.py (#990) Valeriu Predoi

CHAPTER

FIFTYONE

V2.1.0

This release includes

51.1 Bug fixes

- Set unit=1 if anomalies are standardized (#727) bascrezee
- Fix crash for FGOALS-g2 variables without longitude coordinate (#729) Bouwe Andela
- Improve variable alias management (#595) Javier Vegas-Regidor
- Fix area_statistics fx files loading (#798) Javier Vegas-Regidor
- Fix units after derivation (#754) Manuel Schlund

51.2 Documentation

- Update v2.0.0 release notes with final additions (#722) Bouwe Andela
- Update package description in setup.py (#725) Mattia Righi
- Add installation instructions for pip installation (#735) Bouwe Andela
- Improve config-user documentation (#740) Bouwe Andela
- Update the zenodo file with contributors (#807) Valeriu Predoi
- Improve command line run documentation (#721) Javier Vegas-Regidor
- Update the zenodo file with contributors (continued) (#810) Valeriu Predoi

51.3 Improvements

- Reduce size of docker image (#723) Javier Vegas-Regidor
- Add 'test' extra to installation, used by docker development tag (#733) Bouwe Andela
- Correct dockerhub link (#736) Bouwe Andela
- Create action-install-from-pypi.yml (#734) Valeriu Predoi
- Add pre-commit for linting/formatting (#766) Stef Smeets
- Run tests in parallel and when building conda package (#745) Bouwe Andela

- Readable exclude pattern for pre-commit (#770) Stef Smeets
- Github Actions Tests (#732) Valeriu Predoi
- Remove isort setup to fix formatting conflict with yapf (#778) Stef Smeets
- Fix yapf-isort import formatting conflict (Fixes #777) (#784) Stef Smeets
- Sorted output for *esmvaltool recipes list* (#790) Stef Smeets
- Replace vmprof with vprof (#780) Valeriu Predoi
- Update CMIP6 tables to 6.9.32 (#706) Javier Vegas-Regidor
- Default config-user path now set in config-user read function (#791) Javier Vegas-Regidor
- Add custom variable lweGrace (#692) bascrezee
- Create Github Actions workflow to build and deploy on Test PyPi and PyPi (#820) Valeriu Predoi
- Build and publish the esmvalcore package to conda via Github Actions workflow (#825) Valeriu Predoi

51.4 Fixes for datasets

- Fix cmip6 models (#629) npgillett
- Fix siconca variable in EC-Earth3 and EC-Earth3-Veg models in amip simulation (#702) Evgenia Galytska

51.5 Preprocessor

- Move cmor_check_data to early in preprocessing chain (#743) Bouwe Andela
- Add RMS iris analysis operator to statistics preprocessor functions (#747) Pep Cos
- Add surface chlorophyll concentration as a derived variable (#720) sloosvel
- Use dask to reduce memory consumption of extract_levels for masked data (#776) Valeriu Predoi

CHAPTER

FIFTYTWO

V2.0.0

This release includes

52.1 Bug fixes

- Fixed derivation of co2s (#594) Manuel Schlund
- Padding while cropping needs to stay within sane bounds for shapefiles that span the whole Earth (#626) Valeriu Predoi
- Fix concatenation of a single cube (#655) Bouwe Andela
- Fix mask fx dict handling not to fail if empty list in values (#661) Valeriu Predoi
- Preserve metadata during anomalies computation when using iris cubes difference (#652) Valeriu Predoi
- Avoid crashing when there is directory 'esmvaltool' in the current working directory (#672) Valeriu Predoi
- Solve bug in ACCESS1 dataset fix for calendar. (#671) Peter Kalverla
- Fix the syntax for adding multiple ensemble members from the same dataset (#678) SarahAlidoost
- Fix bug that made preprocessor with fx files fail in rare cases (#670) Manuel Schlund
- Add support for string coordinates (#657) Javier Vegas-Regidor
- Fixed the shape extraction to account for wraparound shapefile coords (#319) Valeriu Predoi
- Fixed bug in time weights calculation (#695) Manuel Schlund
- Fix diagnostic filter (#713) Javier Vegas-Regidor

52.2 Documentation

- Add pandas as a requirement for building the documentation (#607) Bouwe Andela
- Document default order in which preprocessor functions are applied (#633) Bouwe Andela
- Add pointers about data loading and CF standards to documentation (#571) Valeriu Predoi
- Config file populated with site-specific data paths examples (#619) Valeriu Predoi
- Update Codacy badges (#643) Bouwe Andela
- Update copyright info on readthedocs (#668) Bouwe Andela
- Updated references to documentation (now docs.esmvaltool.org) (#675) Axel Lauer

- Add all European grants to Zenodo (#680) Bouwe Andela
- Update Sphinx to v3 or later (#683) Bouwe Andela
- Increase version to 2.0.0 and add release notes (#691) Bouwe Andela
- Update setup.py and README.md for use on PyPI (#693) Bouwe Andela
- Suggested Documentation changes (#690) Steve Smith

52.3 Improvements

- Reduce the size of conda package (#606) Bouwe Andela
- Add a few unit tests for DiagnosticTask (#613) Bouwe Andela
- Make ncl or R tests not fail if package not installed (#610) Valeriu Predoi
- Pin flake8<3.8.0 (#623) Valeriu Predoi
- Log warnings for likely errors in provenance record (#592) Bouwe Andela
- Unpin flake8 (#646) Bouwe Andela
- More flexible native6 default DRS (#645) Bouwe Andela
- Try to use the same python for running diagnostics as for esmvaltool (#656) Bouwe Andela
- Fix test for lower python version and add note on lxml (#659) Valeriu Predoi
- Added 1m deep average soil moisture variable (#664) bascrezee
- Update docker recipe (#603) Javier Vegas-Regidor
- Improve command line interface (#605) Javier Vegas-Regidor
- Remove utils directory (#697) Bouwe Andela
- Avoid pytest version that crashes (#707) Bouwe Andela
- Options arg in read_config_user_file now optional (#716) Javier Vegas-Regidor
- Produce a readable warning if ancestors are a string instead of a list. (#711) katjaweigel
- Pin Yamale to v2 (#718) Bouwe Andela
- Expanded cmor public API (#714) Manuel Schlund

52.4 Fixes for datasets

- Added various fixes for hybrid height coordinates (#562) Manuel Schlund
- Extended fix for cl-like variables of CESM2 models (#604) Manuel Schlund
- Added fix to convert "geopotential" to "geopotential height" for ERA5 (#640) Evgenia Galytska
- Do not fix longitude values if they are too far from valid range (#636) Javier Vegas-Regidor

52.5 Preprocessor

- Implemented concatenation of cubes with derived coordinates (#546) Manuel Schlund
- Fix derived variable ctotal calculation depending on project and standard name (#620) Valeriu Predoi
- State of the art FX variables handling without preprocessing (#557) Valeriu Predoi
- Add max, min and std operators to multimodel (#602) Javier Vegas-Regidor
- Added preprocessor to extract amplitude of cycles (#597) Manuel Schlund
- Overhaul concatenation and allow for correct concatenation of multiple overlapping datasets (#615) Valeriu Predoi
- Change volume stats to handle and output masked array result (#618) Valeriu Predoi
- Area_weights for cordex in area_statistics (#631) mwjury
- Accept cubes as input in multimodel (#637) sloosvel
- Make multimodel work correctly with yearly data (#677) Valeriu Predoi
- Optimize time weights in time preprocessor for climate statistics (#684) Valeriu Predoi
- Add percentiles to multi-model stats (#679) Peter Kalverla

CHAPTER

FIFTYTHREE

V2.0.0B9

This release includes

53.1 Bug fixes

- Cast dtype float32 to output from zonal and meridional area preprocessors (#581) Valeriu Predoi

53.2 Improvements

- Unpin on Python<3.8 for conda package (run) (#570) Valeriu Predoi
- Update pytest installation marker (#572) Bouwe Andela
- Remove vmrh2o (#573) Mattia Righi
- Restructure documentation (#575) Bouwe Andela
- Fix mask in land variables for CCSM4 (#579) Klaus Zimmermann
- Fix derive scripts wrt required method (#585) Klaus Zimmermann
- Check coordinates do not have repeated standard names (#558) Javier Vegas-Regidor
- Added derivation script for co2s (#587) Manuel Schlund
- Adapted custom co2s table to match CMIP6 version (#588) Manuel Schlund
- Increase version to v2.0.0b9 (#593) Bouwe Andela
- Add a method to save citation information (#402) SarahAlidoost

For older releases, see the release notes on https://github.com/ESMValGroup/ESMValCore/releases.

Part IX

Indices and tables

- genindex
- search

PYTHON MODULE INDEX

e

esmvalcore.cmor, 179
esmvalcore.cmor.check, 179
esmvalcore.cmor.fix, 184
esmvalcore.cmor.fixes, 187
esmvalcore.cmor.table, 188
esmvalcore.config, 201
esmvalcore.dataset, 205
esmvalcore.esgf.facets, 215
esmvalcore.exceptions, 217
esmvalcore.experimental.recipe, 261
esmvalcore.experimental.recipe_metadata, 271
esmvalcore.experimental.recipe_output, 265
esmvalcore.experimental.utils, 275
esmvalcore.iris_helpers, 219
esmvalcore.local, 221
esmvalcore.preprocessor, 223
esmvalcore.typing, 257

INDEX

A

accumulate_coordinate() (*in module esmvalcore.preprocessor*), 225
add_altitude_from_plev() (*in module esmvalcore.cmor.fixes*), 187
add_leading_dim_to_cube() (*in module esmvalcore.iris_helpers*), 219
add_note() (*esmvalcore.experimental.recipe_metadata.RenderError method*), 273
add_plev_from_altitude() (*in module esmvalcore.cmor.fixes*), 187
add_supplementary() (*esmvalcore.dataset.Dataset method*), 206
add_supplementary_variables() (*in module esmvalcore.preprocessor*), 225
amplitude() (*in module esmvalcore.preprocessor*), 225
annual_statistics() (*in module esmvalcore.preprocessor*), 226
anomalies() (*in module esmvalcore.preprocessor*), 226
area_statistics() (*in module esmvalcore.preprocessor*), 227
args (*esmvalcore.experimental.recipe_metadata.RenderError attribute*), 273
augment_facets() (*esmvalcore.dataset.Dataset method*), 206
authors (*esmvalcore.experimental.recipe_output.DataFile property*), 266
authors (*esmvalcore.experimental.recipe_output.ImageFile property*), 267
authors (*esmvalcore.experimental.recipe_output.OutputFile property*), 268
axis (*esmvalcore.cmor.table.CoordinateInfo attribute*), 191
axis_statistics() (*in module esmvalcore.preprocessor*), 227

B

bias() (*in module esmvalcore.preprocessor*), 228

C

caption (*esmvalcore.experimental.recipe_output.DataFile property*), 266
caption (*esmvalcore.experimental.recipe_output.ImageFile property*), 267
caption (*esmvalcore.experimental.recipe_output.OutputFile property*), 268
CFG (*in module esmvalcore.config*), 201
check_data() (*esmvalcore.cmor.check.CMORCheck method*), 181
check_metadata() (*esmvalcore.cmor.check.CMORCheck method*), 181
CheckLevels (*class in esmvalcore.cmor.check*), 179
citation_file (*esmvalcore.experimental.recipe_output.DataFile property*), 266
citation_file (*esmvalcore.experimental.recipe_output.ImageFile property*), 267
citation_file (*esmvalcore.experimental.recipe_output.OutputFile property*), 268
clear() (*esmvalcore.cmor.table.TableInfo method*), 194
climate_statistics() (*in module esmvalcore.preprocessor*), 228
clip() (*in module esmvalcore.preprocessor*), 229
clip_timerange() (*in module esmvalcore.preprocessor*), 229
CMIP3Info (*class in esmvalcore.cmor.table*), 188
CMIP5Info (*class in esmvalcore.cmor.table*), 189
CMIP6Info (*class in esmvalcore.cmor.table*), 190
cmor_check() (*in module esmvalcore.cmor.check*), 184
cmor_check_data() (*in module esmvalcore.cmor.check*), 183
cmor_check_data() (*in module esmvalcore.preprocessor*), 229
cmor_check_metadata() (*in module esmvalcore.cmor.check*), 183
cmor_check_metadata() (*in module esmvalcore.preprocessor*), 230
CMOR_TABLES (*in module esmvalcore.cmor.table*), 191
CMORCheck (*class in esmvalcore.cmor.check*), 180
CMORCheckError, 180
concatenate() (*in module esmvalcore.preprocessor*), 230

Config (*class in esmvalcore.config*), 201
config_dir (*esmvalcore.config.Session property*), 202
Contributor (*class in esmvalcore.experimental.recipe_metadata*), 271
convert_units() (*in module esmvalcore.preprocessor*), 230
CoordinateInfo (*class in esmvalcore.cmor.table*), 191
coordinates (*esmvalcore.cmor.table.VariableInfo attribute*), 196
copy() (*esmvalcore.cmor.table.TableInfo method*), 195
copy() (*esmvalcore.cmor.table.VariableInfo method*), 196
copy() (*esmvalcore.dataset.Dataset method*), 206
create() (*esmvalcore.experimental.recipe_output.DataFile class method*), 266
create() (*esmvalcore.experimental.recipe_output.ImageFile class method*), 267
create() (*esmvalcore.experimental.recipe_output.OutputFile class method*), 268
create_dataset_map() (*in module esmvalcore.esgf.facets*), 216
CustomInfo (*class in esmvalcore.cmor.table*), 192

D

daily_statistics() (*in module esmvalcore.preprocessor*), 231
data (*esmvalcore.experimental.recipe.Recipe property*), 261
data_citation_file (*esmvalcore.experimental.recipe_output.DataFile property*), 266
data_citation_file (*esmvalcore.experimental.recipe_output.ImageFile property*), 267
data_citation_file (*esmvalcore.experimental.recipe_output.OutputFile property*), 269
data_files (*esmvalcore.experimental.recipe_output.TaskOutput property*), 271
DataFile (*class in esmvalcore.experimental.recipe_output*), 265
Dataset (*class in esmvalcore.dataset*), 205
dataset (*esmvalcore.esgf.ESGFFile attribute*), 214
DATASET_MAP (*in module esmvalcore.esgf.facets*), 215
datasets_to_recipe() (*in module esmvalcore.dataset*), 208
date2num() (*in module esmvalcore.iris_helpers*), 219
DEBUG (*esmvalcore.cmor.check.CheckLevels attribute*), 180
decadal_statistics() (*in module esmvalcore.preprocessor*), 231
DEFAULT (*esmvalcore.cmor.check.CheckLevels attribute*), 180
DEFAULT_ORDER (*in module esmvalcore.preprocessor*), 223
depth_integration() (*in module esmvalcore.preprocessor*), 232
derive() (*in module esmvalcore.preprocessor*), 232
detrend() (*in module esmvalcore.preprocessor*), 232
DiagnosticOutput (*class in esmvalcore.experimental.recipe_output*), 266
diagnostics (*esmvalcore.experimental.recipe_output.RecipeOutput attribute*), 269
dimensions (*esmvalcore.cmor.table.VariableInfo attribute*), 196
download() (*esmvalcore.esgf.ESGFFile method*), 215
download() (*in module esmvalcore.esgf*), 214

E

ensemble_statistics() (*in module esmvalcore.preprocessor*), 233
Error, 217
ESGFFile (*class in esmvalcore.esgf*), 214
esmvalcore.cmor
    module, 179
esmvalcore.cmor.check
    module, 179
esmvalcore.cmor.fix
    module, 184
esmvalcore.cmor.fixes
    module, 187
esmvalcore.cmor.table
    module, 188
esmvalcore.config
    module, 201
esmvalcore.dataset
    module, 205
esmvalcore.esgf.facets
    module, 215
esmvalcore.exceptions
    module, 217
esmvalcore.experimental.recipe
    module, 261
esmvalcore.experimental.recipe_metadata
    module, 271
esmvalcore.experimental.recipe_output
    module, 265
esmvalcore.experimental.utils
    module, 275
esmvalcore.iris_helpers
    module, 219
esmvalcore.local
    module, 221
esmvalcore.preprocessor
    module, 223
esmvalcore.typing

module, 257
ESMValCoreDeprecationWarning, 217
extract_coordinate_points() (*in module esmvalcore.preprocessor*), 233
extract_levels() (*in module esmvalcore.preprocessor*), 234
extract_location() (*in module esmvalcore.preprocessor*), 234
extract_month() (*in module esmvalcore.preprocessor*), 235
extract_named_regions() (*in module esmvalcore.preprocessor*), 235
extract_point() (*in module esmvalcore.preprocessor*), 236
extract_region() (*in module esmvalcore.preprocessor*), 237
extract_season() (*in module esmvalcore.preprocessor*), 237
extract_shape() (*in module esmvalcore.preprocessor*), 237
extract_time() (*in module esmvalcore.preprocessor*), 238
extract_trajectory() (*in module esmvalcore.preprocessor*), 239
extract_transect() (*in module esmvalcore.preprocessor*), 239
extract_volume() (*in module esmvalcore.preprocessor*), 240

F

facets (*esmvalcore.dataset.Dataset attribute*), 205
facets (*esmvalcore.esgf.ESGFFile attribute*), 214
facets (*esmvalcore.local.LocalFile property*), 221
FACETS (*in module esmvalcore.esgf.facets*), 216
Facets (*in module esmvalcore.typing*), 257
FacetValue (*in module esmvalcore.typing*), 257
files (*esmvalcore.dataset.Dataset property*), 206
FILTER_ATTRS (*esmvalcore.experimental.recipe_output.RecipeOutput attribute*), 270
find() (*esmvalcore.experimental.utils.RecipeList method*), 275
find_files() (*esmvalcore.dataset.Dataset method*), 207
find_files() (*in module esmvalcore.esgf*), 211
find_files() (*in module esmvalcore.local*), 221
fix_data() (*in module esmvalcore.cmor.fix*), 184
fix_data() (*in module esmvalcore.preprocessor*), 240
fix_file() (*in module esmvalcore.cmor.fix*), 185
fix_file() (*in module esmvalcore.preprocessor*), 241
fix_metadata() (*in module esmvalcore.cmor.fix*), 186
fix_metadata() (*in module esmvalcore.preprocessor*), 241
frequency (*esmvalcore.cmor.check.CMORCheck attribute*), 180
frequency (*esmvalcore.cmor.table.VariableInfo attribute*), 196
from_core_recipe_output() (*esmvalcore.experimental.recipe_output.RecipeOutput class method*), 270
from_dict() (*esmvalcore.experimental.recipe_metadata.Contributor class method*), 272
from_files() (*esmvalcore.dataset.Dataset method*), 207
from_ranges() (*esmvalcore.dataset.Dataset method*), 207
from_recipe() (*esmvalcore.dataset.Dataset static method*), 207
from_tag() (*esmvalcore.experimental.recipe_metadata.Contributor class method*), 272
from_tag() (*esmvalcore.experimental.recipe_metadata.Project class method*), 272
from_tag() (*esmvalcore.experimental.recipe_metadata.Reference class method*), 273
from_task() (*esmvalcore.experimental.recipe_output.TaskOutput class method*), 271
fromkeys() (*esmvalcore.cmor.table.TableInfo method*), 195

G

generic_lev_name (*esmvalcore.cmor.table.CoordinateInfo attribute*), 191
get() (*esmvalcore.cmor.table.TableInfo method*), 195
get() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270
get_all_recipes() (*in module esmvalcore.experimental.utils*), 275
get_next_month() (*in module esmvalcore.cmor.fixes*), 187
get_output() (*esmvalcore.experimental.recipe.Recipe method*), 261
get_recipe() (*in module esmvalcore.experimental.utils*), 275
get_table() (*esmvalcore.cmor.table.CMIP3Info method*), 188
get_table() (*esmvalcore.cmor.table.CMIP5Info method*), 189
get_table() (*esmvalcore.cmor.table.CMIP6Info method*), 190
get_table() (*esmvalcore.cmor.table.CustomInfo method*), 192
get_table() (*esmvalcore.cmor.table.InfoBase method*), 193

get_time_bounds() (*in module esmvalcore.cmor.fixes*), 187
get_var_info() (*in module esmvalcore.cmor.table*), 197
get_variable() (*esmvalcore.cmor.table.CMIP3Info method*), 189
get_variable() (*esmvalcore.cmor.table.CMIP5Info method*), 189
get_variable() (*esmvalcore.cmor.table.CMIP6Info method*), 190
get_variable() (*esmvalcore.cmor.table.CustomInfo method*), 193
get_variable() (*esmvalcore.cmor.table.InfoBase method*), 193

H

has_coord_with_standard_name() (*esmvalcore.cmor.table.VariableInfo method*), 196
has_debug_messages() (*esmvalcore.cmor.check.CMORCheck method*), 182
has_errors() (*esmvalcore.cmor.check.CMORCheck method*), 182
has_warnings() (*esmvalcore.cmor.check.CMORCheck method*), 182
hourly_statistics() (*in module esmvalcore.preprocessor*), 242

I

IGNORE (*esmvalcore.cmor.check.CheckLevels attribute*), 180
image_files (*esmvalcore.experimental.recipe_output.TaskOutput property*), 271
ImageFile (*class in esmvalcore.experimental.recipe_output*), 266
info (*esmvalcore.experimental.recipe_output.RecipeOutput attribute*), 269
InfoBase (*class in esmvalcore.cmor.table*), 193
INHERITED_FACETS (*in module esmvalcore.dataset*), 208
InputFilesNotFound, 217
InvalidConfigParameter, 217
items() (*esmvalcore.cmor.table.TableInfo method*), 195
items() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270

J

JsonInfo (*class in esmvalcore.cmor.table*), 194

K

keys() (*esmvalcore.cmor.table.TableInfo method*), 195
keys() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270
kind (*esmvalcore.experimental.recipe_output.DataFile attribute*), 266
kind (*esmvalcore.experimental.recipe_output.ImageFile attribute*), 267
kind (*esmvalcore.experimental.recipe_output.OutputFile attribute*), 269

L

linear_trend() (*in module esmvalcore.preprocessor*), 242
linear_trend_stderr() (*in module esmvalcore.preprocessor*), 243
load() (*esmvalcore.dataset.Dataset method*), 208
load() (*in module esmvalcore.preprocessor*), 243
load_from_file() (*esmvalcore.config.Config method*), 201
load_iris() (*esmvalcore.experimental.recipe_output.DataFile method*), 266
load_xarray() (*esmvalcore.experimental.recipe_output.DataFile method*), 266
local_file() (*esmvalcore.esgf.ESGFFile method*), 215
LocalFile (*class in esmvalcore.local*), 221
long_name (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
long_name (*esmvalcore.cmor.table.VariableInfo attribute*), 196

M

main_log (*esmvalcore.config.Session property*), 202
main_log_debug (*esmvalcore.config.Session property*), 202
mask_above_threshold() (*in module esmvalcore.preprocessor*), 243
mask_below_threshold() (*in module esmvalcore.preprocessor*), 244
mask_fillvalues() (*in module esmvalcore.preprocessor*), 244
mask_glaciated() (*in module esmvalcore.preprocessor*), 245
mask_inside_range() (*in module esmvalcore.preprocessor*), 245
mask_landsea() (*in module esmvalcore.preprocessor*), 245
mask_landseaice() (*in module esmvalcore.preprocessor*), 246
mask_multimodel() (*in module esmvalcore.preprocessor*), 246
mask_outside_range() (*in module esmvalcore.preprocessor*), 246
merge_cube_attributes() (*in module esmvalcore.iris_helpers*), 220
meridional_statistics() (*in module esmvalcore.preprocessor*), 246

minimal_facets (*esmvalcore.dataset.Dataset property*), 208
MissingConfigParameter, 217
modeling_realm (*esmvalcore.cmor.table.VariableInfo attribute*), 196
module
    esmvalcore.cmor, 179
    esmvalcore.cmor.check, 179
    esmvalcore.cmor.fix, 184
    esmvalcore.cmor.fixes, 187
    esmvalcore.cmor.table, 188
    esmvalcore.config, 201
    esmvalcore.dataset, 205
    esmvalcore.esgf.facets, 215
    esmvalcore.exceptions, 217
    esmvalcore.experimental.recipe, 261
    esmvalcore.experimental.recipe_metadata, 271
    esmvalcore.experimental.recipe_output, 265
    esmvalcore.experimental.utils, 275
    esmvalcore.iris_helpers, 219
    esmvalcore.local, 221
    esmvalcore.preprocessor, 223
    esmvalcore.typing, 257
monthly_statistics() (*in module esmvalcore.preprocessor*), 247
multi_model_statistics() (*in module esmvalcore.preprocessor*), 247
must_have_bounds (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192

N

name (*esmvalcore.esgf.ESGFFile attribute*), 214
name (*esmvalcore.experimental.recipe.Recipe property*), 261
NetCDFAttr (*in module esmvalcore.typing*), 257

O

out_name (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
OutputFile (*class in esmvalcore.experimental.recipe_output*), 268

P

plot_dir (*esmvalcore.config.Session property*), 202
pop() (*esmvalcore.cmor.table.TableInfo method*), 195
popitem() (*esmvalcore.cmor.table.TableInfo method*), 195
positive (*esmvalcore.cmor.table.VariableInfo attribute*), 196
preproc_dir (*esmvalcore.config.Session property*), 202
Project (*class in esmvalcore.experimental.recipe_metadata*), 272
provenance_xml_file (*esmvalcore.experimental.recipe_output.DataFile property*), 266
provenance_xml_file (*esmvalcore.experimental.recipe_output.ImageFile property*), 267
provenance_xml_file (*esmvalcore.experimental.recipe_output.OutputFile property*), 269

R

read_cmor_tables() (*in module esmvalcore.cmor.table*), 197
read_json() (*esmvalcore.cmor.table.CoordinateInfo method*), 192
read_json() (*esmvalcore.cmor.table.VariableInfo method*), 196
read_main_log() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270
read_main_log_debug() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270
Recipe (*class in esmvalcore.experimental.recipe*), 261
RecipeError, 217
RecipeList (*class in esmvalcore.experimental.utils*), 275
RecipeOutput (*class in esmvalcore.experimental.recipe_output*), 269
Reference (*class in esmvalcore.experimental.recipe_metadata*), 272
references (*esmvalcore.experimental.recipe_output.DataFile property*), 266
references (*esmvalcore.experimental.recipe_output.ImageFile property*), 268
references (*esmvalcore.experimental.recipe_output.OutputFile property*), 269
regrid() (*in module esmvalcore.preprocessor*), 249
regrid_time() (*in module esmvalcore.preprocessor*), 250
relative_main_log (*esmvalcore.config.Session attribute*), 202
relative_main_log_debug (*esmvalcore.config.Session attribute*), 202
relative_plot_dir (*esmvalcore.config.Session attribute*), 202
relative_preproc_dir (*esmvalcore.config.Session attribute*), 202
relative_run_dir (*esmvalcore.config.Session attribute*), 202
relative_work_dir (*esmvalcore.config.Session attribute*), 203

RELAXED (*esmvalcore.cmor.check.CheckLevels attribute*), 180
reload() (*esmvalcore.config.Config method*), 201
remove_supplementary_variables() (*in module esmvalcore.preprocessor*), 250
render() (*esmvalcore.experimental.recipe.Recipe method*), 261
render() (*esmvalcore.experimental.recipe_metadata.Reference method*), 273
render() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270
RenderError, 273
report() (*esmvalcore.cmor.check.CMORCheck method*), 182
report_critical() (*esmvalcore.cmor.check.CMORCheck method*), 182
report_debug_message() (*esmvalcore.cmor.check.CMORCheck method*), 183
report_debug_messages() (*esmvalcore.cmor.check.CMORCheck method*), 182
report_error() (*esmvalcore.cmor.check.CMORCheck method*), 182
report_errors() (*esmvalcore.cmor.check.CMORCheck method*), 181
report_warning() (*esmvalcore.cmor.check.CMORCheck method*), 183
report_warnings() (*esmvalcore.cmor.check.CMORCheck method*), 182
requested (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
resample_hours() (*in module esmvalcore.preprocessor*), 250
resample_time() (*in module esmvalcore.preprocessor*), 251
rolling_window_statistics() (*in module esmvalcore.preprocessor*), 251
run() (*esmvalcore.experimental.recipe.Recipe method*), 261
run_dir (*esmvalcore.config.Session property*), 203

S

save() (*in module esmvalcore.preprocessor*), 252
seasonal_statistics() (*in module esmvalcore.preprocessor*), 252
Session (*class in esmvalcore.config*), 201
session (*esmvalcore.dataset.Dataset property*), 208
session (*esmvalcore.experimental.recipe_output.RecipeOutput attribute*), 269
session_dir (*esmvalcore.config.Session property*), 203
session_name (*esmvalcore.config.Session attribute*), 203
set_facet() (*esmvalcore.dataset.Dataset method*), 208
set_session_name() (*esmvalcore.config.Session method*), 203
set_version() (*esmvalcore.dataset.Dataset method*), 208
setdefault() (*esmvalcore.cmor.table.TableInfo method*), 195
short_name (*esmvalcore.cmor.table.VariableInfo attribute*), 196
size (*esmvalcore.esgf.ESGFFile attribute*), 214
standard_name (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
standard_name (*esmvalcore.cmor.table.VariableInfo attribute*), 196
start_session() (*esmvalcore.config.Config method*), 201
stored_direction (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
STRICT (*esmvalcore.cmor.check.CheckLevels attribute*), 180
summary() (*esmvalcore.dataset.Dataset method*), 208
supplementaries (*esmvalcore.dataset.Dataset attribute*), 205
SuppressedError, 217

T

TableInfo (*class in esmvalcore.cmor.table*), 194
TaskOutput (*class in esmvalcore.experimental.recipe_output*), 270
timeseries_filter() (*in module esmvalcore.preprocessor*), 253
to_base64() (*esmvalcore.experimental.recipe_output.ImageFile method*), 268

U

units (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
units (*esmvalcore.cmor.table.VariableInfo attribute*), 197
update() (*esmvalcore.cmor.table.TableInfo method*), 195
urls (*esmvalcore.esgf.ESGFFile attribute*), 214

V

valid_max (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
valid_max (*esmvalcore.cmor.table.VariableInfo attribute*), 197
valid_min (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
valid_min (*esmvalcore.cmor.table.VariableInfo attribute*), 197
value (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192

values() (*esmvalcore.cmor.table.TableInfo method*), 195
values() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270
var_name (*esmvalcore.cmor.table.CoordinateInfo attribute*), 192
VariableInfo (*class in esmvalcore.cmor.table*), 195
volume_statistics() (*in module esmvalcore.preprocessor*), 253

W

weighting_landsea_fraction() (*in module esmvalcore.preprocessor*), 254
with_traceback() (*esmvalcore.experimental.recipe_metadata.RenderError method*), 273
work_dir (*esmvalcore.config.Session property*), 203
write_html() (*esmvalcore.experimental.recipe_output.RecipeOutput method*), 270

Z

zonal_statistics() (*in module esmvalcore.preprocessor*), 254