ScriptEngine Tasks EC-Earth

Release 0.1

Nov 17, 2022

Getting Started

1 Installation **3**
- 1.1 Installation using conda and pip . . . 3
- 1.2 Installation from Source . . . 4

2 Using the Taskset **5**

3 General Structure **7**

4 Generic Processing Tasks **9**
- 4.1 Scalar . . . 9
- 4.2 Timeseries . . . 9
- 4.3 LinearCombination . . . 11

5 Computational Performance Diagnostics **13**
- 5.1 DiskusageRteScalar . . . 13
- 5.2 SimulatedyearsRteScalar . . . 13
- 5.3 SYPD using Timeseries . . . 14

6 NEMO Diagnostics **15**
- 6.1 NemoGlobalMeanYearMeanTimeseries . . . 15
- 6.2 NemoAllMeanMap . . . 16
- 6.3 NemoYearMeanTemporalMap . . . 16
- 6.4 NemoMonthMeanTemporalMap . . . 17

7 SI3 Diagnostics **19**
- 7.1 Si3HemisSumMonthMeanTimeseries . . . 19
- 7.2 Si3HemisPointMonthMeanAllMeanMap . . . 20
- 7.3 Si3HemisPointMonthMeanTemporalmap . . . 20

8 OpenIFS Diagnostics **23**
- 8.1 OifsGlobalMeanYearMeanTimeseries . . . 23
- 8.2 OifsAllMeanMap . . . 24
- 8.3 OifsYearMeanTemporalmap . . . 24

9 Presentation Tasks **25**
- 9.1 Markdown . . . 25
- 9.2 Redmine . . . 26

10 Developer's Guide **27**
10.1 General Notes for Contributing 27
10.2 Code Structure 27
10.3 Logging Policy 28
10.4 Naming Processing Tasks 28
10.5 Naming Presentation Tasks 29

This package contains ScriptEngine tasks for the EC-Earth 4 climate model, specifically the EC-Earth 4 monitoring tool.

CHAPTER 1

Installation

The ScriptEngine tasks for EC-Earth require a Unix(-like) operating system and Python 3.6+.

The package supports two different installation methods:

- Installation using conda and PyPI/pip;
- Installation from source.

The next sections will detail the procedure to install the package for both methods.

You can check if everything worked out by calling `se --help` from the command line. ScriptEngine will show all registered tasks, and the tasks in this package start with the prefix `ece.`.

1.1 Installation using conda and pip

This is the recommended way to install this package.

Get the conda package manager following the instructions for your operating system. Create an environment and activate it with

```
conda create --name your_environment_name python=3.6 # or 3.7 or 3.8
conda activate your_environment_name
```

Alternatively, activate the existing conda environment you want to use for this package.

Update your conda environment using the file `conda_environment.yml` in the GitHub repository:

```
conda env update -n your_environment_name --file conda_environment.yml
```

This YAML file contains necessary dependencies for packages that *should* be installed via conda (like, e.g., Iris). You can also install these packages from source, but this will require a lot more attention during the setup process. Refer to the documentation of the packages in `conda_environment.yml` for more information on installing them.

The ScriptEngine tasks for EC-Earth can then be installed using

```
pip install scriptengine-tasks-ecearth
```

The remaining dependencies will be installed automatically.

1.2 Installation from Source

You can download or clone the source code from https://github.com/uwefladrich/scriptengine-tasks-ecearth.

Update your conda environment using the file `conda_environment.yml` in the GitHub repository:

```
conda env update -n your_environment_name --file conda_environment.yml
```

This YAML file contains necessary dependencies for packages that *should* be installed via conda (like, e.g., Iris). You can also install these packages from source, but this will require a lot more attention during the setup process. Refer to the documentation of the packages in `conda_environment.yml` for more information on installing them.

The package can be installed from inside the *scriptengine-tasks-ecearth* directory (assuming you did not choose a different name) using

```
pip install -e .
```

If you want to run the tests, you will need to download the test data, put the files into `tests/testdata`, and install Pytest. You can run them from inside the *scriptengine-tasks-ecearth* directory using

```
pytest .
```

To build the documentation manually, you will need Sphinx. The HTML theme is the Read the Docs Sphinx Theme.

CHAPTER 2

Using the Taskset

To run a YAML script using the ScriptEngine tasks for EC-Earth, use the following command:

```
se example_script.yml
```

where `example_script.yml` contains tasks from this package.

See `se --help` for additional commands such as setting the logging level etc.

For more information on how to use ScriptEngine in general, please consult the ScriptEngine documentation.

CHAPTER 3

General Structure

The monitoring tool has a defined workflow: At the end of each leg, using EC-Earth output, it creates relevant diagnostics. These diagnostics are then visualized in an expressive manner. Both the selection of diagnostics as well as the desired visualization might vary between experiments and are thus configurable. The software consists of two different task types:

Processing tasks process input from the model output and the runtime environment. With this, they create diagnostics and save them in a file, the diagnostic on disk. The tool so far supports four types of diagnostics:

- scalar: zero-dimensional in time & space.
- time series: zero-dimensional in space, one-dimensional in time.
- map: two-dimensional in space, zero-dimensional in time.
- temporal map: two-dimensional in space, zero-dimensional in time.

Processing tasks and the resulting diagnostics on disk should be named according to the naming scheme described here: *Naming Processing Tasks*.

Presentation tasks read these saved diagnostics and visualize them. Then, they present all diagnostics at a presentation outlet.

CHAPTER 4

Generic Processing Tasks

The processing tasks in this chapter are not tied to a specific EC-Earth component.

4.1 Scalar

Diagnostic Type: Scalar
Mapped to: `ece.mon.scalar`

This is the base class for all implemented scalar tasks. The Scalar processing task writes custom output to a YAML file.

Required arguments

- `title`: Title of the diagnostic
- `value`: Value of the scalar
- `dst`: Destination, must end in *.yml* or *.yaml*

Optional arguments

- `comment`: Additional description of diagnostic. Default: `None`.

```
- ece.mon.scalar:
    title: "Experiment ID"
    value: "{{exp_id}}"
    dst: "{{mondir}}/expid_scalar.yml"
```

4.2 Timeseries

Diagnostic Type: Timeseries
Mapped to: `ece.mon.timeseries`

This processing task creates a time series diagnostic, illustrating the progression of a scalar quantity over the duration of the current experiment. It can be used for custom output, an exemplary use case is the SYPD time series, a computational performance diagnostic: *SYPD using Timeseries*.

Required arguments

- `title`: Title of the diagnostic
- `data_value`: Value of the new data point
- `coord_value`: New value of the time coordinate (can be int/float/double or date/datetime). Must be monotonically increasing.
- `dst`: Destination, must end in *.nc*

Optional arguments

- `comment`: Additional description of diagnostic. Default: "."
- `data_name`: Name of the data variable. Default: value of `title`
- `data_unit`: Unit of the data variable. Can be one of the UDUNITS strings. Default: 1
- `coord_name`: Name of the coordinate. Default: *time*
- `coord_unit`: Unit of the coordinate. Can be one of the UDUNITS strings. Default: 1

Note: To check if a unit string is compatible with UDUNITS, use the following small Python check:

```
import cf_units
cf_units.as_unit("kg") # insert your test string here
```

This will throw a *ValueError* in case the unit is not compatible with UDUNITS.

4.2.1 Minimal Example

```
- ece.mon.timeseries:
    title: "Some Diagnostic"
    data_value: "{{some_value}}"
    coord_value: "{{leg_num}}"
    dst: "{{mondir}}/diagnostic_timeseries.nc"
```

4.2.2 Elaborate Example

```
- ece.mon.timeseries:
    title: "An Interesting Title"
    data_value: "{{some_value}}"
    coord_value: "{{some_other_value}}"
    dst: "{{mondir}}/diagnostic_timeseries.nc"
    comment: "Diagnostic Description."
    coord_name: "x-axis label"
    coord_units: "s"
    data_name: "y-axis label"
    data_units: "m"
```

4.3 LinearCombination

Diagnostic Type: None (generates a general NetCDF files for further processing tasks)
Mapped to: `ece.mon.linear_combination`

This processing task computes a linear combination $\sum \alpha_i v_i$ of scalar factors α_i and compatible NetCDF variables v_i, and writes the result into a new NetCDF file. This can be used to create custom diagnostics, for example the difference or (weighted) sum of variables. The resulting NetCDF files and variables can be used in subsequent processing tasks. Note that the `LinearCombination` task does not perform any spatial or temporal averaging.

Example use cases for `LinearCombination` include the computation of precipitation minus evaporation (mass balance $P - E$) and energy balances by summing up individual radiation contributions.

The input variables for the `LinearCombination` task must be compatible in terms of dimensions and units, according to the rules of Iris cube maths. If Iris cannot compute the linear combination, the execution of the task is aborted with an error explaining the type of incompatibility.

Required arguments

- `src`: A list of dictionaries, each containing the `path`, `varname`, for each variable, and an optional scalar `factor` α_i. The default `factor` is 1.0.
- `dst`: A dictionary describing the NetCDF file used to store the custom diagnostic. Must at least contain the `path` and `varname` for the result.

Optional ‘‘dst‘‘ arguments

- `longname`: The *long name* of the target variable. If not provided, the resulting diagnostic will not have a longname.
- `standardname`: A *valid standard name* for the target variable as defined by the *CF conventions*. If not provided, the resulting diagnostic will not have a standardname.
- `unit`: Custom target unit for the destination file. Can be one of the UDUNITS strings (see the *above note on checking valid units*). If not provided, Iris will try to determine the unit of the linear combination.

4.3.1 Examples

The first example adds short and long wave radiation contributions to provide an energy budget at the top of atmosphere (TOA):

```
- ece.mon.linear_combination:
    src:
      - varname: rsnt
        path: oifs_output_file.nc
      - varname: rlnt
        path: oifs_output_file.nc
    dst:
      varname: net_toa
      longname: Net TOA
      path: net_toa.nc
```

The second example computes the difference between precipitation P and evaporation E to provide the mass balance $P-E$ as custom diagnostic in `pme.nc`. Note that the standardname and unit are explicitely set for the output NetCDF file:

```
- ece.mon.linear_combination:
    src:
      - varname: pr
        path: oifs_output_file.nc
      - varname: evspsbl
        path: oifs_output_file.nc
        factor: -1.0
    dst:
      varname: pme
      longname: "Precipitation - Evaporation"
      standardname: precipitation_amount
      unit: "kg m-2"
      path: pme.nc
```

CHAPTER 5

Computational Performance Diagnostics

The processing tasks in this chapter create diagnostics informing about computational performance and the general experiment progress.

5.1 DiskusageRteScalar

Diagnostic Type: Scalar
Mapped to: `ece.mon.diskusage_rte_scalar`

Computes the size of a user-specified directory.

Required arguments

- `src`: Path to the specified directory.
- `dst`: Destination, must end in *.yml* or *.yaml*

```
- ece.mon.diskusage_rte_scalar:
    src: "{{rundir}}/output"
    dst: "{{mondir}}/diskusage_rte_scalar.yml"
```

5.2 SimulatedyearsRteScalar

Diagnostic Type: Scalar
Mapped to: `ece.mon.simulatedyears_rte_scalar`

Computes the difference in years between `end` and `start`.

Required arguments

- `start`: Start date of the simulation.
- `end`: End date of the current leg.
- `dst`: Destination, must end in *.yml* or *.yaml*

```
- ece.mon.simulatedyears_rte_scalar:
    start: "{{start}}"
    end: "{{leg.end}}"
    dst: "{{mondir}}/simulatedyears_rte_scalar.yml"
```

5.3 SYPD using Timeseries

```
- ece.mon.timeseries:
    title: "Simulated Years per Day"
    coord_value: "{{leg_num}}"
    coord_name: "Leg Number"
    comment: "SYPD development during this simulation."
    data_value: "{{((schedule.leg.end - schedule.leg.start)/script_elapsed_time/365)}}
↪"
    dst: "{{mondir}}/sypd_timeseries.nc"
```

CHAPTER 6

NEMO Diagnostics

The processing tasks in this chapter create diagnostics for the NEMO ocean model.

Assumptions about input data

- input data are output files from NEMO, i.e. NetCDF files on a global curvilinear grid.
- currently, only 2D variables can be treated.
- it is assumed that data for land cells is flagged as invalid.
- A leg length of one year is expected. Longer/shorter lengths won't lead to failure but file descriptions might be inaccurate (e.g. the *comment* attribute might say "annual mean" despite being a half-year mean).

6.1 NemoGlobalMeanYearMeanTimeseries

Diagnostic Type: Time Series
Mapped to: `ece.mon.nemo_global_mean_year_mean_timeseries`

This processing task computes the global and temporal average of a 2D oceanic quantity, resulting in a time series diagnostic.

To compute an annual mean, the leg has to be one year long. If it is, e.g., six months long, the task will compute the six month global mean of the input variable.

Required arguments

- `src`: A list of strings containing paths to the desired NEMO output files. This list can be manually entered or (often better) created by the `find` task.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.
- `domain`: A string containing the path to the `domain.nc` file. Used to compute the global mean.
- `varname`: The name of the oceanic variable as it is saved in the NEMO output file.

Optional arguments

- `grid`: The grid type of the desired variable. Can be T, U, V, W. Default: T.

```
- ece.mon.nemo_global_mean_year_mean_timeseries:
    src: "{{t_files}}"
    dst: "{{mondir}}/tos_nemo_global_mean_year_mean_timeseries.nc"
    domain: "{{rundir}}/domain.nc"
    varname: tos
```

6.2 NemoAllMeanMap

Diagnostic Type: Map
Map Type: global ocean
Mapped to: `ece.mon.nemo_all_mean_map`

This task takes the "simulation average climatology" (i.e., a multi-year mean) of a global 2D ocean variable and saves it as a map diagnostic on disk.

Required arguments

- `src`: A list of strings containing paths to the desired NEMO output files. This list can be manually entered or (often better) created by the `find` task.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.
- `varname`: The name of the oceanic variable as it is saved in the NEMO output file.

```
- ece.mon.nemo_all_mean_map:
    src: "{{t_files}}"
    dst: "{{mondir}}/tos_nemo_all_mean_map.nc"
    varname: "tos"
```

6.3 NemoYearMeanTemporalMap

Diagnostic Type: Temporal Map
Map Type: global ocean
Mapped to: `ece.mon.nemo_year_mean_temporalmap`

This task takes the leg mean of a global 2D ocean variable and saves it as a temporal map diagnostic on disk. It assumes the leg is one year long, which is why it is called "YearMeanTemporalMap".

Required arguments

- `src`: A list of strings containing paths to the desired NEMO output files. This list can be manually entered or (often better) created by the `find` task.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.
- `varname`: The name of the oceanic variable as it is saved in the NEMO output file.

```
- ece.mon.nemo_year_mean_temporalmap:
    src: "{{t_files}}"
    dst: "{{mondir}}/tos_nemo_year_mean_temporalmap.nc"
    varname: "tos"
```

6.4 NemoMonthMeanTemporalMap

Diagnostic Type: Temporal Map
Map Type: global ocean
Mapped to: `ece.mon.nemo_month_mean_temporalmap`

Saves consecutive monthly mean maps of a global 2D ocean variable as a temporal map. This task will fail if the output frequency is not monthly (e.g. daily or annual output).

Required arguments

- `src`: A list of strings containing paths to the desired NEMO output files. This list can be manually entered or (often better) created by the `find` task.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.
- `varname`: The name of the oceanic variable as it is saved in the NEMO output file.

```
- ece.mon.nemo_month_mean_temporalmap:
    src: "{{t_files}}"
    dst: "{{mondir}}/tos_nemo_month_mean_temporalmap.nc"
    varname: "tos"
```

CHAPTER 7

SI3 Diagnostics

The processing tasks in this chapter create diagnostics for the SI3 sea ice model.

Assumptions about input data:

- input data are output files from SI3, i.e. NetCDF files on a global curvilinear grid.
- data for land cells is flagged as invalid.
- monthly output files. Otherwise the tasks will fail to compute the diagnostics.

7.1 Si3HemisSumMonthMeanTimeseries

Diagnostic Type: Time Series
Mapped to: `ece.mon.si3_hemis_sum_month_mean_timeseries`

Computes the hemispheric sum of a sea ice variable's month mean, resulting in a time series diagnostic. This can be used to create a seasonal cycle time series (March-September or similar) or the time series of another selection of months.

Required arguments

- `src`: A list of strings or a single string containing paths to the desired SI3 output file(s).
- `domain`: A string containing the path to the `domain.nc` file. The variables `e1t` and `e2t` are used for computing the area weights.
- `varname`: The name of the ice variable as saved in the output file. Can be `sivolu` or `siconc`.
- `hemisphere`: The name of the requested hemisphere. Can be `north` or `south`.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.

```
- ece.mon.si3_hemis_sum_month_mean_timeseries:
    src:
        - "{{feb_file[0]}}"
        - "{{sep_file[0]}}"
    domain: "{{rundir}}/domain_cfg.nc"
    dst: "{{mondir}}/siarea_si3_south_sum_feb+sep_mean_timeseries.nc"
    hemisphere: south
    varname: siconc
```

7.2 Si3HemisPointMonthMeanAllMeanMap

Diagnostic Type: Map
Mapped to: `ece.mon.si3_hemis_point_month_mean_all_mean_map`
Map Type: polar ice sheet

Computes the simulation average climatology of a sea ice variable's month mean on one hemisphere, resulting in a map diagnostic. E.g. the simulation mean of all March means of the arctic sea ice concentration.

Required arguments

- `src`: A string containing paths to the desired SI3 output file.
- `varname`: The name of the ice variable as saved in the output file. Can be `sivolu` or `siconc`.
- `hemisphere`: The name of the requested hemisphere. Can be `north` or `south`.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.

```
- ece.mon.si3_hemis_point_month_mean_all_mean_map:
    src: "{{ice_file_sep}}"
    dst: "{{mondir}}/sivolu_si3_north_point_sep_mean_all_mean_map.nc"
    hemisphere: south
    varname: sivolu
```

7.3 Si3HemisPointMonthMeanTemporalmap

Diagnostic Type: Temporal Map
Mapped to: `ece.mon.si3_hemis_point_month_mean_temporalmap`
Map Type: polar ice sheet

Creates a temporal map of a sea ice variable's month mean, resulting in a temporal map diagnostic. E.g. the March means of the arctic sea ice concentration over time.

Required arguments

- `src`: A string containing paths to the desired SI3 output file.
- `varname`: The name of the ice variable as saved in the output file. Can be `sivolu` or `siconc`.
- `hemisphere`: The name of the requested hemisphere. Can be `north` or `south`.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.

```
- ece.mon.si3_hemis_point_month_mean_temporalmap:
        src: "{{ice_file_mar}}"
        dst: "{{mondir}}/siconc_si3_north_point_mar_mean_temporalmap.nc"
        hemisphere: north
        varname: siconc
```

CHAPTER 8

OpenIFS Diagnostics

These processing tasks assume that the provided input file is a NetCDF output file from OpenIFS with monthly data. Further assumptions:

- regularly-spaced in time
- data in grid-point space on a reduced Gaussian grid
- 2D variables only

8.1 OifsGlobalMeanYearMeanTimeseries

Diagnostic Type: Time Series
Mapped to: `ece.mon.oifs_global_mean_year_mean_timeseries`

This processing task computes the global and temporal average of a 2D atmospheric quantity, resulting in a time series diagnostic.

To compute an annual mean, the leg has to be one year long. If it is, e.g., six months long, the task will compute the six month global mean of the input variable.

Required arguments

- `src`: A string containing the path to the OpenIFS output file.
- `varname`: The name of the variable in the output file. Refer to the ECMWF parameter database for the meaning of the variables.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.

```
- ece.mon.oifs_global_mean_year_mean_timeseries:
    src: "{{rundir}}/output/oifs/{{exp_id}}_atm_1m_1990-1990.nc"
    varname: 2t
    dst: "{{mondir}}/2t_oifs_global_mean_year_mean_timeseries.nc"
```

8.2 OifsAllMeanMap

Diagnostic Type: Map
Map Type: global atmosphere
Mapped to: `ece.mon.oifs_all_mean_map`

This task takes the "simulation average climatology" (i.e., a multi-year mean) of a global 2D atmospheric variable and saves it as a map diagnostic on disk.

Required arguments

- `src`: A string containing the path to the OpenIFS output file.
- `varname`: The name of the variable in the output file. Refer to the ECMWF parameter database for the meaning of the variables.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.

```
- ece.mon.oifs_all_mean_map:
    src: "{{rundir}}/output/oifs/{{exp_id}}_atm_1m_1990-1990.nc"
    varname: 2t
    dst: "{{mondir}}/2t_oifs_all_mean_map.nc"
```

8.3 OifsYearMeanTemporalmap

Diagnostic Type: Temporal Map
Map Type: global atmosphere
Mapped to: `ece.mon.oifs_year_mean_temporalmap`

This task takes the leg mean of a global 2D ocean variable and saves it as a temporal map diagnostic on disk. It assumes the leg is one year long, which is why it is called "YearMeanTemporalMap".

Required arguments

- `src`: A string containing the path to the OpenIFS output file.
- `varname`: The name of the variable in the output file. Refer to the ECMWF parameter database for the meaning of the variables.
- `dst`: A string ending in `.nc`. This is where the diagnostic will be saved.

```
- ece.mon.oifs_year_mean_temporalmap:
    src: "{{rundir}}/output/oifs/{{exp_id}}_atm_1m_1990-1990.nc"
    varname: 2t
    dst: "{{mondir}}/2t_oifs_year_mean_temporalmap.nc"
```

CHAPTER 9

Presentation Tasks

9.1 Markdown

Mapped to `ece.mon.presentation.markdown`.

This presentation task creates a Markdown file which contains visualizations of the created diagnostics on disk.

Required arguments

- `src`: A list of strings containing paths to the diagnostics on disk that should be presented.
- `dst`: A string containing the path to the directory where the report should be put. The directory will contain the image files for the presentation, as well as a file `summary.md` with the final presentation.
- `template`: A string containing the path to the Markdown template file. An exemplary file is contained in the `docs/template` folder in the repository.

```
- ece.mon.presentation.markdown:
    src:
        - "{{mondir}}/description.yml"
        - "{{mondir}}/exp-id.yml"
        - "{{mondir}}/output-disk-usage.yml"
        - "{{mondir}}/tos-global-avg.nc"
        - "{{mondir}}/sos-global-avg.nc"
    dst: "{{mondir}}/report"
    template: "scriptengine-tasks-ecearth/docs/templates/markdown_template.md.j2"
```

9.1.1 Custom Visualization Options

For custom visualization, a dictionary instead of the path alone can be passed as a source. The path then must lie at the key `path`. Currently, the following customization features are implemented:

- `value_range`: set the minimum and maximum value of a time series or (temporal) map. Particularly useful for temporal maps. Default: `[None, None]`

- `colormap`: set a custom colormap for maps and temporal maps. Default: `RdBu_r`. The list of possible colormaps is in the Matplotlib documentation.

Example:

```
- ece.mon.presentation.markdown
    src:
        - "{{mondir}}/description.yml"
        - "{{mondir}}/exp-id.yml"
        - "{{mondir}}/output-disk-usage.yml"
        - path: "{{mondir}}/tos_nemo_global_mean_year_mean_timeseries.nc"
          value_range: [13, 17]
        - path: "{{mondir}}/tos_nemo_year_mean_temporalmap.nc"
          value_range: [-2, 30]
          colormap: 'viridis'
    dst: "{{mondir}}/report"
    template: "scriptengine-tasks-ecearth/docs/templates/markdown_template.md.j2"
```

9.2 Redmine

Mapped to `ece.mon.presentation.redmine`.

This presentation task creates a Redmine issue on the EC-Earth development portal, containing visualizations of the created diagnostics on disk.

Required arguments

- `src`: A list of strings containing paths to the diagnostics on disk that should be presented. You can use the *Custom Visualization Options* in the same way as with the Markdown task.
- `local_dst`: A string containing the path to the directory where the attachments can be stored locally. The directory will contain the image files for the presentation, as well as a file `issue_description.txt` with the issue description text.
- `template`: A string containing the path to the issue description template file. An exemplary file is contained in the `docs/template` folder in the repository.
- `api_key`: Your API key for logging in to the EC-Earth development portal. You can find it (you might have to generate it first) in your account settings.
- `subject`: The name of your issue. A recommended format for this is shown below.

```
- ece.mon.presentation.redmine:
    src:
        - "{{mondir}}/description.yml"
        - "{{mondir}}/exp-id.yml"
        - "{{mondir}}/sim-years.yml"
        - "{{mondir}}/tos-global-avg.nc"
        - "{{mondir}}/sos-global-avg.nc"
        - "{{mondir}}/sithic-north-mar.nc"
        - "{{mondir}}/sithic-north-sep.nc"
    local_dst: "{{mondir}}/redmine-report"
    api_key: # Your API key for the EC-Earth Dev Portal
    subject: "{{exp_id}}: Short Description"
    template: "scriptengine-tasks-ecearth/docs/templates/redmine_template.txt.j2"
```

CHAPTER 10

Developer's Guide

This part summarizes guidelines for extending the monitoring tool.

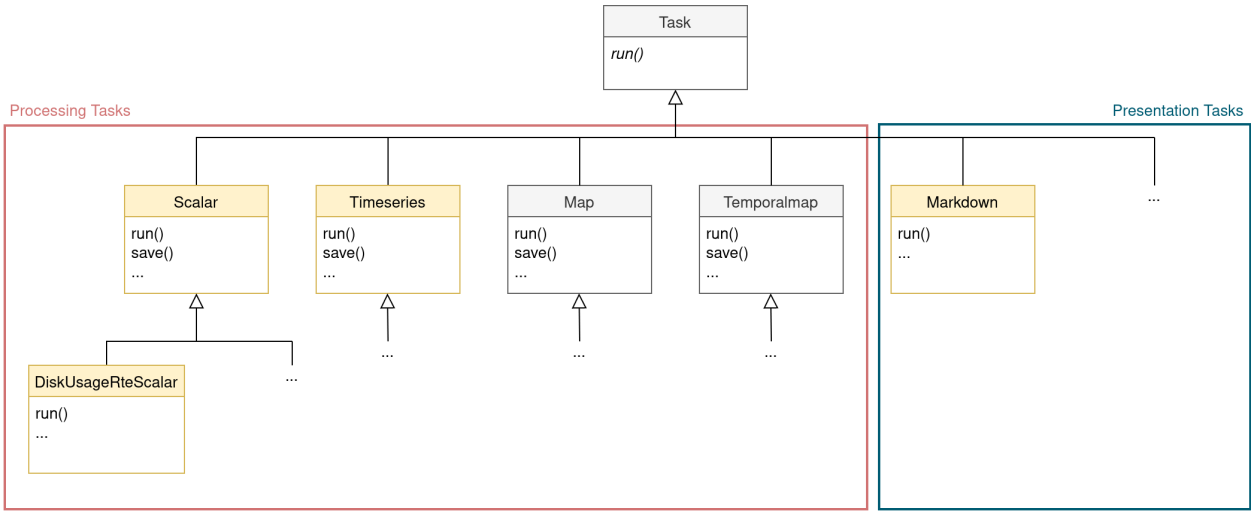
10.1 General Notes for Contributing

Use black and isort to format the code you add.

10.2 Code Structure

All monitoring tasks are contained in the folder monitoring. helpers contains modules with functions and classes that are needed in multiple unrelated, processing and presentation tasks.

All monitoring tasks inherit from the generic ScriptEngine Task. Processing tasks have an inheritance substructure defined by their diagnostic type. New processing tasks should adhere to these structure: Either inherit from the corresponding Diagnostictype class (Scalar, Timeseries, Map, Temporalmap) or create a new one if it is a new diagnostic type. The Diagnostictype class provides a `save()` method that can be reused by all inherited tasks. This structure is illustrated below:



To make a task accessible to users, add it as an entry point to setup.py.

10.3 Logging Policy

Every (monitoring) task is responsible for logging its execution. In the beginning of `self.run()` (after *very few* lines of code), a task **must** call `self.log_info()` to log that it is active. It may include information about its arguments in the log message, but nothing that would require a lot of code processing the arguments (which could fail). A task **should** write only one `log_info` message during execution. General "progression" statements **must** be `log_debug` messages. If a monitoring task experiences an unrecoverable error: It **must** use `log_error` and throw one of the ScriptEngine Task Exceptions. For problems which do not lead to a `ScriptEngineTaskException`, use `log_warning` instead.

10.4 Naming Processing Tasks

A processing task has a name that appears in multiple places:

- the Python class
- the Python module
- the YAML representation
- the diagnostics on disk created by it

These all adhere to the same naming scheme: `variable_component_[domain_op_...]_diagnostictype`. The Python class uses the name in CamelCase naming convention. YAML representation, module, and diagnostic on disk use the snake_case naming convention (see the usage example).

Component	Spatial Domain	Spatial Operation	Temporal Domain	Temporal Operation	Diagnostic Type
oifs	global	mean	*season*: djf, mam,...	mean	scalar
nemo	*hemis*: north, south	sum	*month*: jan, feb,...	min	timeseries
si3	*region*: region names	min	year	max	map
lpjg		max	all		temporalmap
fesom		point?			
xios		...			
oasis					
si3					
rte					

Italic keywords can be used as placeholders for the keywords they describe. If a user can select the operation of the domain, use *op* as a placeholder.

The `domain_op` combination can be used consecutively, e.g.: `global_sum_month_max_year_mean`. The variable keyword is less standardized, e.g. amocstrength, sypd, tos, 2t, 167,... Depending on the diagnostic/processing task, parts of the naming scheme are unnecessary. `diagnostictype` may not be omitted.

10.4.1 Usage Example: Naming Scheme

- Python class: `NemoGlobalMeanYearMeanTimeseries`
- Python module: `nemo_global_mean_year_mean_timeseries`
- YAML representation: `ece.nemo_global_mean_year_mean_timeseries`
- the diagnostics on disk created by it: `ece.mon.tos_nemo_global_mean_year_mean_timeseries.nc`

10.5 Naming Presentation Tasks

Naming presentation tasks is not as standardized as for processing tasks. The task/class/module name should be the presentation outlet, e.g. Markdown. Their YAML representation is preceded by *ece.mon.presentation* to make them distinguishable from processing tasks.