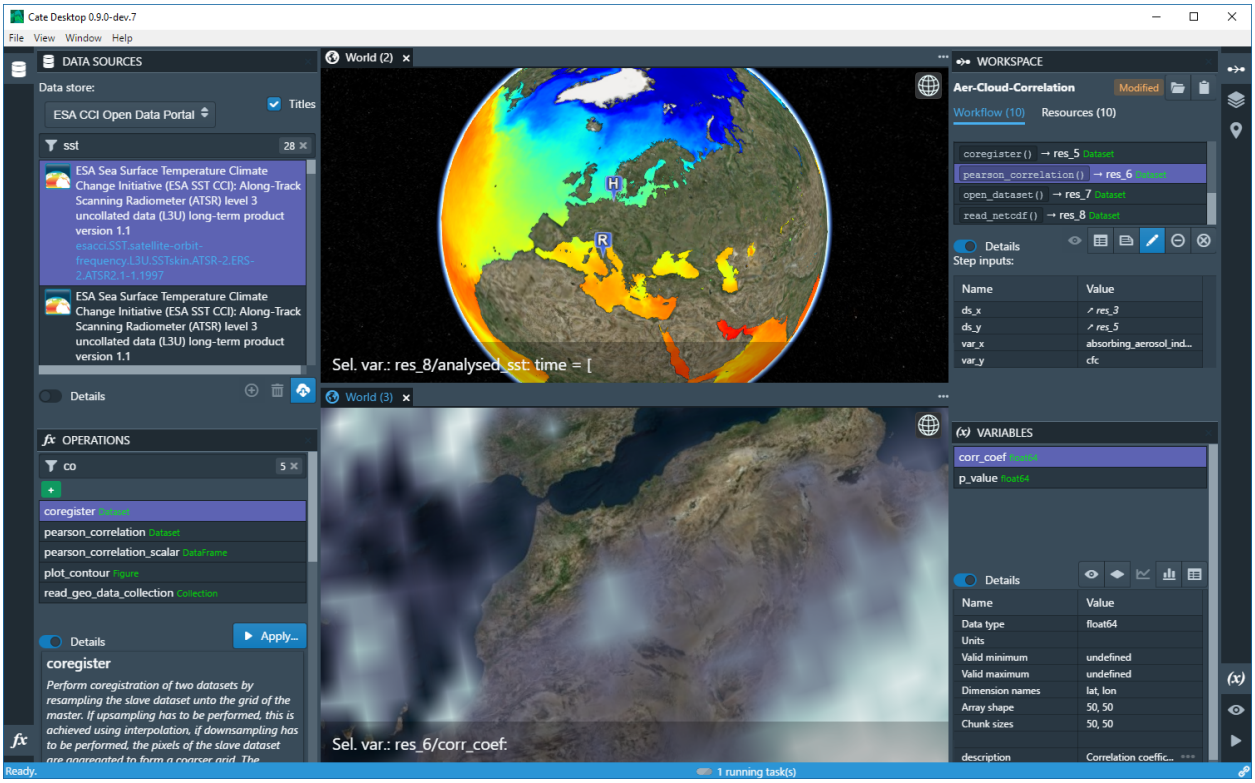
ESA CCI Toolbox Documentation

Release 2.0.0.dev15

cate Development Team

Jun 18, 2018

Table of Contents


Cate Desktop 0.9.0-dev.7
File View Window Help
DATA SOURCES
Data store:
ESA CCI Open Data Portal
Titles
sst
28
ESA Sea Surface Temperature Climate Change Initiative (ESA SST CCI): Along-Track Scanning Radiometer (ATSR) level 3 uncollated data (L3U) long-term product version 1.1
esacci.SST.satellite-orbit-frequency.L3U.SSTskin.ATSR-2.ERS-2.ATSR2.1-1.1997
ESA Sea Surface Temperature Climate Change Initiative (ESA SST CCI): Along-Track Scanning Radiometer (ATSR) level 3 uncollated data (L3U) long-term product version 1.1
Details
OPERATIONS
co
5
coregister Dataset
pearson_correlation Dataset
pearson_correlation_scalar DataFrame
plot_contour Figure
read_geo_data_collection Collection
Details
Apply...
coregister
Perform coregistration of two datasets by resampling the slave dataset unto the grid of the master. If upsampling has to be performed, this is achieved using interpolation, if downsampling has to be performed, the pixels of the slave dataset
World (2)
Sel. var.: res_8/analysed_sst: time = [
World (3)
Sel. var.: res_6/corr_coef:
WORKSPACE
Aer-Cloud-Correlation
Modified
Workflow (10)
Resources (10)
coregister() → res_5 Dataset
pearson_correlation() → res_6 Dataset
open_dataset() → res_7 Dataset
read_netcdf() → res_8 Dataset
Details
Step inputs:
Name
Value
ds_x
res_3
ds_y
res_5
var_x
absorbing_aerosol_ind...
var_y
cfc
VARIABLES
corr_coef float64
p_value float64
Details
Name
Value
Data type
float64
Units
Valid minimum
undefined
Valid maximum
undefined
Dimension names
lat, lon
Array shape
50, 50
Chunk sizes
50, 50
description
Correlation coeffic...
Ready.
1 running task(s)

CHAPTER 1

Introduction

1.1 Project Background

In 2009, ESA, the European Space Agency, has launched the Climate Change Initiative (CCI), a programme to respond the need for climate-quality satellite data as expressed by GCOS, the Global Climate Observing System that supports the UNFCCC, the United Nations Framework Convention on Climate Change.

In the ESA CCI programme **14 Essential Climate Variables** (ECV) are produced by individual expert teams, and cross-cutting activities provide coordination, harmonisation and support. The **CCI Toolbox** and the **CCI Open Data Portal** are the two main technical support projects within the programme. The CCI Open Data Portal will provide a single point of harmonised access to a subset of mature and validated ECV-related data products. The CCI Toolbox will provide tools that support visualisation, analysis and processing across CCI and other climate data products. With these two technical cross cutting activities ESA is providing an interface between its CCI projects and the ECVs generated there, and the wider climate change user community (see following Fig. 1.1).

1.2 Key Objectives

In this context the four key objectives of the CCI Toolbox are:

- Provide to climate users an intuitive software application that is capable of **ingesting data from all CCI projects** and synergistically use this data in a uniform tooling environment. This requires the application to abstract from the various data types used to represent the different ECVs (vector data, n-D raster data), and from data formats used (NetCDF, Shapefiles), and also from various data sources (remote services, local files). A **Common Data Model** (CDM) shall be developed and utilised that can represent all data of all ECVs and make it available to a variety of algorithms and converters to different representations.
- Provide to users a rich set of **data processing operations** that implement commonly used climate algorithms and which operate solely on the CDM. A processor may generate a new CDM instance as output which again is independent from any particular external representation. Processors that use a CDM as input and output can be used to build processing chains that represent typical climate workflows.
- Provide to users various **visualisation and analysis** operations. Again, these functions will solely operate on the CDM. The majority of visualisation and analysis functions will be applicable to multiple ECVs (e.g. 3D

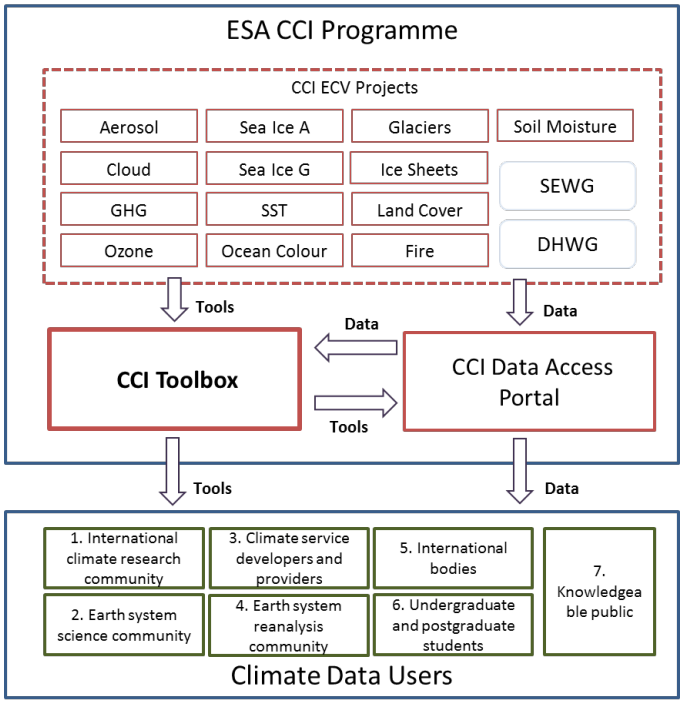


Fig. 1.1: CCI Toolbox context

visualisations) while other may only work if the CDM fulfils certain constraints (e.g. profile data). Some of these functions may generate a new or enrich an existing CDM instance and thus be implemented as processors and be used in processing chains.

- Design the **architecture** of the CCI Toolbox so that it **can be extended by new climate operations** and that it also allows for **reuse of existing or planned software tools and libraries**. Furthermore allow other scientists and tool developers to use the underlying CCI Toolbox algorithms and libraries in their own programs. Scientists are used to stay with their preferred programming language, so ideally the CCI Toolbox shall offer application programming interfaces (API) for programming languages commonly used in climate science such as Python and R, C and FORTRAN.

CHAPTER 2

About Cate

The name *Cate* stands for "Climate Analysis Toolbox for ESA" and is a software developed to facilitate processing and analysis of all the data products generated by the ESA Climate Change Initiative Programme (CCI).

2.1 Interfaces

Cate comprises three major software interfaces:

1. **Cate Desktop** is a user-friendly desktop application. It provides a graphical user interface to all the functionality provided by the CLI. In addition, the GUI allows for visualising data on 3D globes and 2D maps.
2. The Cate **Command-Line Interface** (CLI) used to access and process data through a command shell or console terminal. Almost all Cate functionality is accessible through its CLI. You may decide to use the CLI if you don't like programming in Python. The CLI may also be used to write batch processing scripts for automation.
3. The Cate **Python API** allows you using Cate functions in your Python programs. Cate is programmed in Python 3.

2.2 Concepts

The Cate software is based on a few simple concepts, which are referred to in all user interfaces. Therefore you should make yourself familiar with them before using Cate.

2.2.1 Data Stores

By default, Cate uses the CCI Open Data Portal (ODP) **remote data store** which provides access to all published CCI datasets. There is also a **local data store**, which is used to synchronise remote data or to add any other data sources to Cate[1].

[1] Currently, only NetCDF files can be used as local data sources. In future releases, we will support other formats such as ESRI Shapefiles and GeoTIFF.

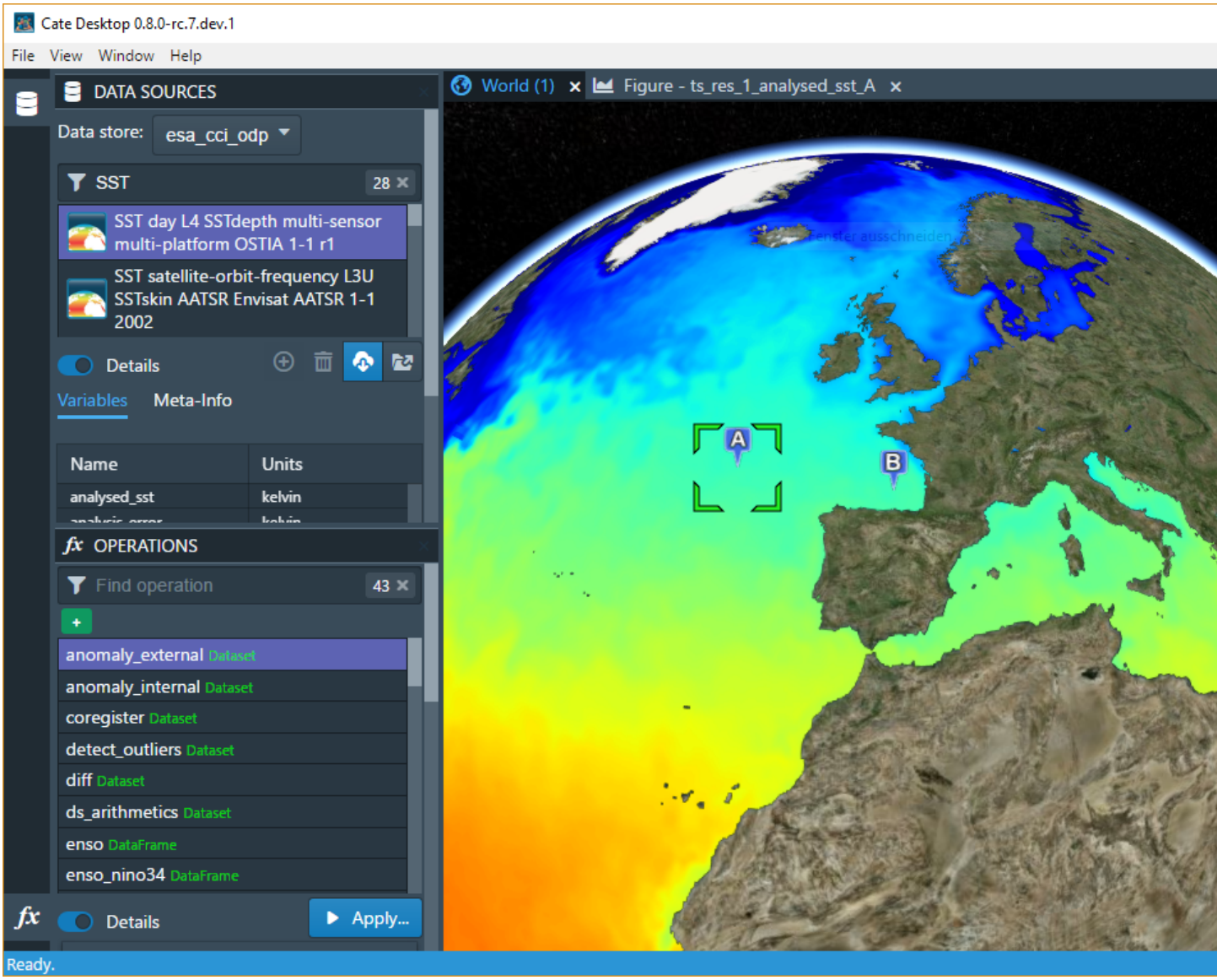


Fig. 2.1: Cate Desktop, this is GUI of the CCI Toolbox

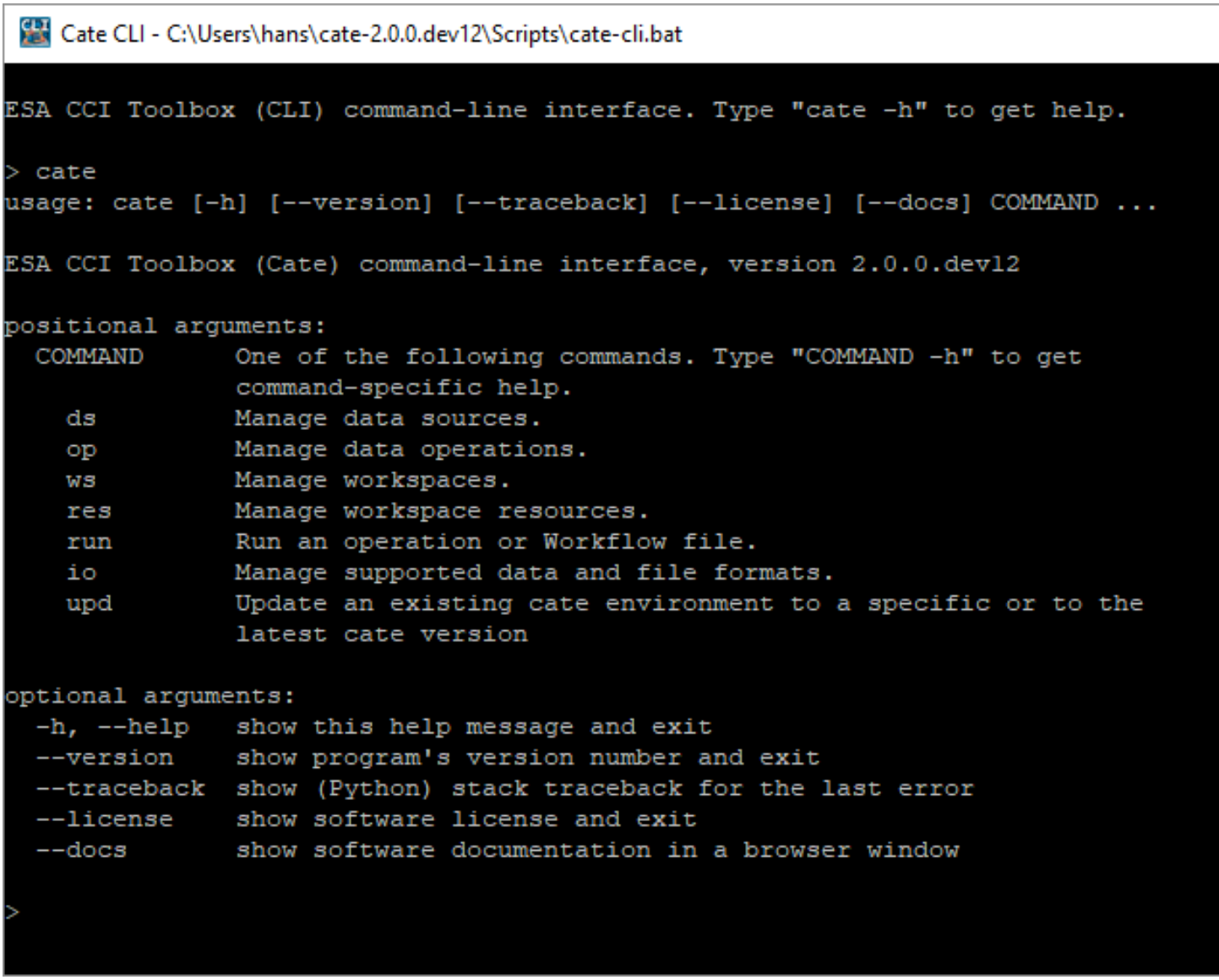


Fig. 2.2: Cate CLI, this is the CLI of the CCI Toolbox

2.2.2 Data Sources

A data store comprises multiple **data sources** which know each dataset's unique identifier and other descriptive information about the dataset. Each data source also knows about the available data access protocols, which may be direct file access, file download via HTTP, or access through OPeNDAP, or a Web Coverage Service (WCS).

In Cate's CLI, `cate ds` is used to perform numerous dataset-related tasks. Type:

```
$ cate ds --help
```

to get an overview of the supported sub-comands.

For example, use:

```
$ cate ds list
```

to list available data sources.

In the GUI, the panel **DATA SOURCES** lets you query and open available data sources.

Note that all remote CCI data source identifiers are prefixed by "esacci.", for example `esacci.SST.day.L4.SSTdepth.multi-sensor.multi-platform.OSTIA.1-0.r1`. Local data source identifiers are prefixed by "local.", for example `local.SST_NAC_2010`.

2.2.3 Datasets

You may **open datasets** from a data source just by providing the dataset's identifier. The underlying physical file structure or access protocol remains transparent. That way, Cate can also deal with datasets that don't fit into your computer's memory, Cate allows for *out-of-core* and *multi-core* processing. However, you can always **read datasets** directly from your local. e.g. NetCDF files or ESRI Shapefiles.

For Python programmers: it might be interesting for you that Cate does not invent new data structures for representing datasets in memory. Instead, opened datasets are represented by data structures defined by the popular Python packages xarray, pandas, and geopandas:

- Gridded and raster datasets (based on NetCDF/CF or OPeNDAP) are represented by *xarray.Dataset* objects[2]. Dataset variables are represented by NumPy-compatible *xarray.DataArray* objects.
- Vector datasets (from ESRI Shapefiles, GeoJSON files) are represented by *geopandas.GeoDataFrame* objects. Dataset variables are represented by pandas-compatible *geopandas.GeoSeries* objects.
- Tabular data (from CSV, Excel files) are represented by *pandas.DataFrame* objects.

2.2.4 Functions and Operations

Cate provides numerous I/O, analysis, and processing **operations** that address typical climate analyses. They are available through all Cate interfaces, the Python API, the CLI, and the GUI.

For Python programmers: Theses *operations* are usual Python functions. The only difference is that Cate has an operation registry where functions to be published for use through the CLI and GUI are registered. In addition to operations provided by Cate, the Python packages xarray, pandas, and geopandas provide a rich and powerful low-level data processing interface for the datasets opened through Cate.

In Cate's CLI, `cate op` is used to perform numerous operation-related tasks. Type:

[2] Currently, only NetCDF and OPeNDAP sources can be represented by `xarray.Dataset` objects. In future releases, we will support other generic formats such as GeoTIFF or HDF.

```
$ cate op --help
```

to get an overview of the supported sub-commands. For example, use:

```
$ cate op list
```

to list and query available operation.

In the GUI, the panel OPERATIONS lets you query and apply all available operations. Applying an operation creates a new *workflow* step in the current *workspace*.

2.2.5 Workflows, Resources, and Workspaces

Using both the CLI and the GUI, users can work in interactive mode, which means that one command creates a certain state which provides a context for another command. In Cate, this can be done without actually storing any data to disk in-between two commands. For example the simple **workflow**

1. open dataset ds1
2. open dataset ds2
3. get variable v1 of ds1
4. get variable v2 of ds2
5. compute v2b which is v2 on the same grid as v1
6. compute c which is the correlation between v1 and v2b
7. output c

can be both executed the same way in the CLI and the GUI. Each step generates a new **resource**, e.g. `ds1`, `v2`. which can serve as input for a subsequent step. Only in the last step, data processing is actually triggered through the workflow, effectively computing and outputting the current value of resource `c`. Currently, Cate workflow steps must refer to a Cate *operation*. Later versions of Cate will also support the following step types:

- Python expressions with access to Cate Python API, xarray, pandas, geopandas, etc.
- Python scripts with access to Cate Python API, xarray, pandas, geopandas, etc.
- Any shell executables
- Other workflows

Workflows are also saved and reopened as part of a Cate **workspace**. A Cate workspace refers to a directory in the user's file system containing a `.cate-workspace` sub-directory, where Cate stores workspace-specific data such as the workspace's workflow. The workflow is saved as a JSON file within that sub-directory together with any other files serving as input or output for the workflow. Relative file paths used as operation parameters are resolved against the current workspace directory. If a workspace is closed, all of its in-memory resources are closed and released.

The following figure `about_workspace_fig` shows the workspace with its contained workflow steps and the associated in-memory resource objects.

In Cate's CLI, you'll find all workspace- and resource-related commands by using the `cate ws` and `cate res` commands:

```
$ cate ws --help
$ cate res --help
```

Using the CLI run command, workflows can be directly executed when given as a JSON-formatted text file:

```
$ cate run <my-workflow.json>
```

More on workflows and its file format can be found in a dedicated chapter workflows.

In Cate's GUI, workspace commands are available in the *File* menu. Furthermore

- the panel WORKSPACE lists all available workspace resources and workflow steps, and
- the panel VARIABLES lists the variables of a selected workspace resource.

Both provide additional workspace-related commands.

CHAPTER 3

Quick Start

This section provides a quick start into Cate by demonstrating how a particular climate use case is performed.

Refer to the *User Manual* for installing the Cate.

The use case describes a climate scientist wishing to analyse potential correlations between the geophysical quantities *Ozone Mole Content* and *Cloud Coverage* in a certain region (see use case description for *Relationships between Aerosol and Cloud ECV*). It requires the toolbox to do the following:

- Access to and ingestion of ESA CCI Ozone and Cloud data (Atmosphere Mole Content of Ozone and Cloud Cover)
- Geometric adjustments (coregistration)
- Spatial (point, polygon) and temporal subsetting
- Visualisation of time series
- Correlation analysis, scatter-plot of correlation statistics, saving of image and correlation statistics

3.1 Using the CLI

In the following, a demonstration is given how the use case described above is performed using the Cate's *Command-Line Interface*.

3.1.1 Dataset Ingestion

Use `ds list` to list available products. You can filter them according to some name.

```
$ cate ds list -n ozone
3 data sources found
   1: esacci.OZONE.day.L3S.TC.GOME-2.Metop-A.MERGED.fv0100.r1
   2: esacci.OZONE.day.L3S.TC.GOME.ERS-2.MERGED.fv0100.r1
   3: esacci.OZONE.mon.L3.NP.multi-sensor.multi-platform.MERGED.fv0002.r1
```

```
$ cate ds list -n cloud
14 data sources found
   1: esacci.CLOUD.day.L3U.CLD_PRODUCTS.AVHRR.NOAA-15.AVHRR_NOAA.1-0.r1
   2: esacci.CLOUD.day.L3U.CLD_PRODUCTS.AVHRR.NOAA-16.AVHRR_NOAA.1-0.r1
   3: esacci.CLOUD.day.L3U.CLD_PRODUCTS.AVHRR.NOAA-17.AVHRR_NOAA.1-0.r1
   4: esacci.CLOUD.day.L3U.CLD_PRODUCTS.AVHRR.NOAA-18.AVHRR_NOAA.1-0.r1
   5: esacci.CLOUD.day.L3U.CLD_PRODUCTS.MODIS.Aqua.MODIS_AQUA.1-0.r1
   6: esacci.CLOUD.day.L3U.CLD_PRODUCTS.MODIS.Terra.MODIS_TERRA.1-0.r1
   7: esacci.CLOUD.mon.L3C.CLD_PRODUCTS.AVHRR.NOAA-15.AVHRR_NOAA.1-0.r1
   8: esacci.CLOUD.mon.L3C.CLD_PRODUCTS.AVHRR.NOAA-16.AVHRR_NOAA.1-0.r1
   9: esacci.CLOUD.mon.L3C.CLD_PRODUCTS.AVHRR.NOAA-17.AVHRR_NOAA.1-0.r1
  10: esacci.CLOUD.mon.L3C.CLD_PRODUCTS.AVHRR.NOAA-18.AVHRR_NOAA.1-0.r1
  11: esacci.CLOUD.mon.L3C.CLD_PRODUCTS.MODIS.Aqua.MODIS_AQUA.1-0.r1
  12: esacci.CLOUD.mon.L3C.CLD_PRODUCTS.MODIS.Terra.MODIS_TERRA.1-0.r1
  13: esacci.CLOUD.mon.L3S.CLD_PRODUCTS.AVHRR.multi-platform.AVHRR_MERGED.1-0.r1
  14: esacci.CLOUD.mon.L3S.CLD_PRODUCTS.MODIS.multi-platform.MODIS_MERGED.1-0.r1
```

Create a new workspace.

```
$ cate ws new
cate-webapi: started service, listening on localhost:49836
Workspace created.
```

Open the desired datasets, by providing their name and desired time-span.

```
$ cate res open cl07 esacci.CLOUD.mon.L3C.CLD_PRODUCTS.multi-sensor.multi-platform.
↪ATSR2-AATSR.2-0.r1
Opening dataset: done
Resource "cl07" set.
```

```
$ cate res open oz07 esacci.OZONE.mon.L3.NP.multi-sensor.multi-platform.MERGED.fv0002.
↪r1 2007-01-01 2007-12-30
Opening dataset: done
Resource "oz07" set.
```

3.1.2 Dataset Variable Selection

To select particular geophysical quantities to work with, use the `select_var` operation together with `cate res set` command:

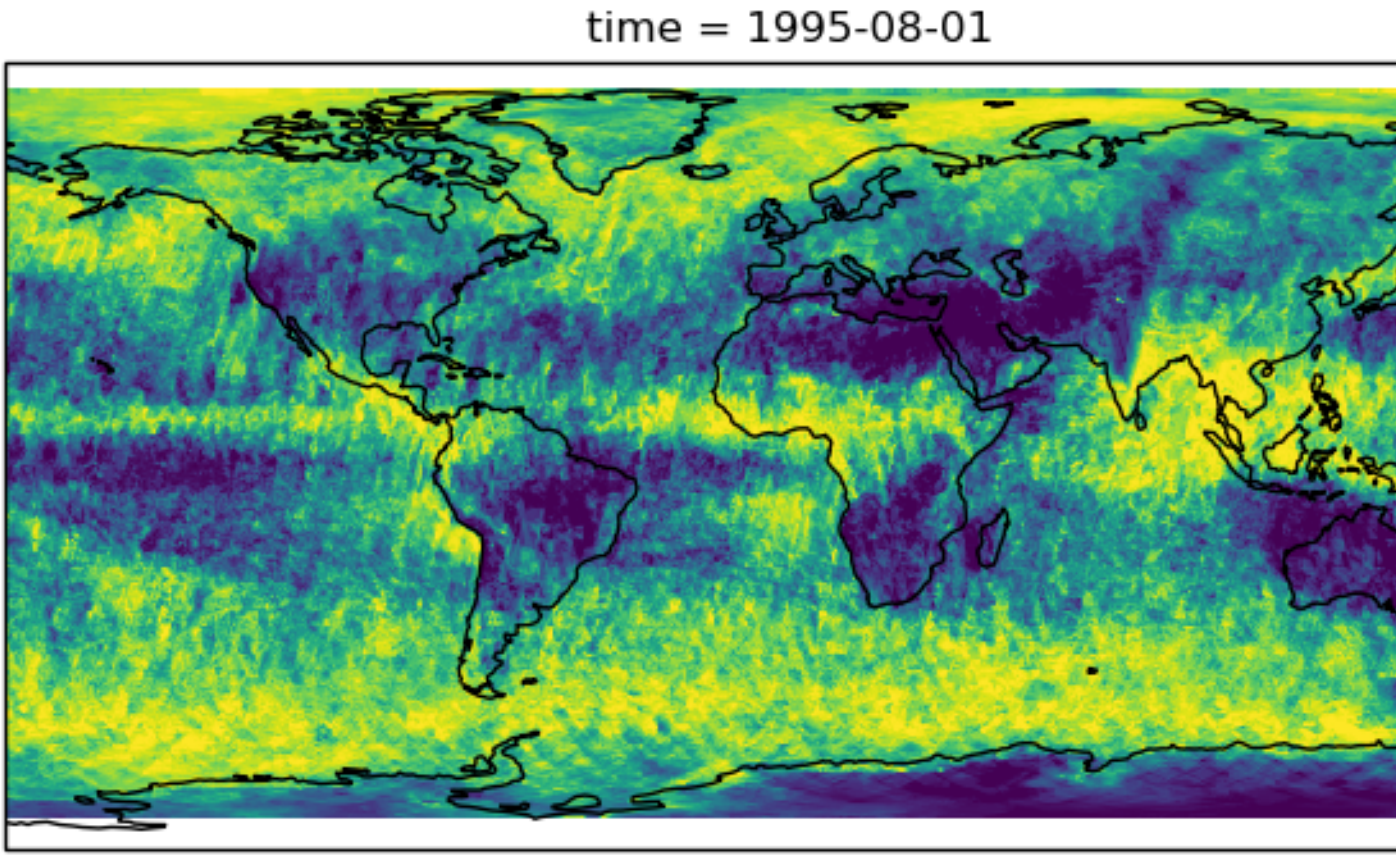
```
$ cate res set cfc select_var ds=@cl07 var=cfc
Executing 2 workflow step(s): done
Resource "cfc" set.
```

```
$ cate res set oz_tot select_var ds=@oz07 var=O3_du_tot
Executing 2 workflow step(s): done
Resource "oz_tot" set.
```

Note the at-character "@" in `@cl07` and `@oz07`. This indicates that the input `ds` of the `select_var` operation will be the output of the respective `open` steps. This establishes a permanent connection between step `open` and `select_var`. In fact, this is the way processing graphs are constructed using the Cate CLI.

We can plot the datasets and save the plots using the `plot_map` operation:

```
$ cate ws run plot_map ds=@cfc var=cfc file=fig1.png
Running operation 'plot_map': Executing 4 workflow step(s)
Operation 'plot_map' executed.
```



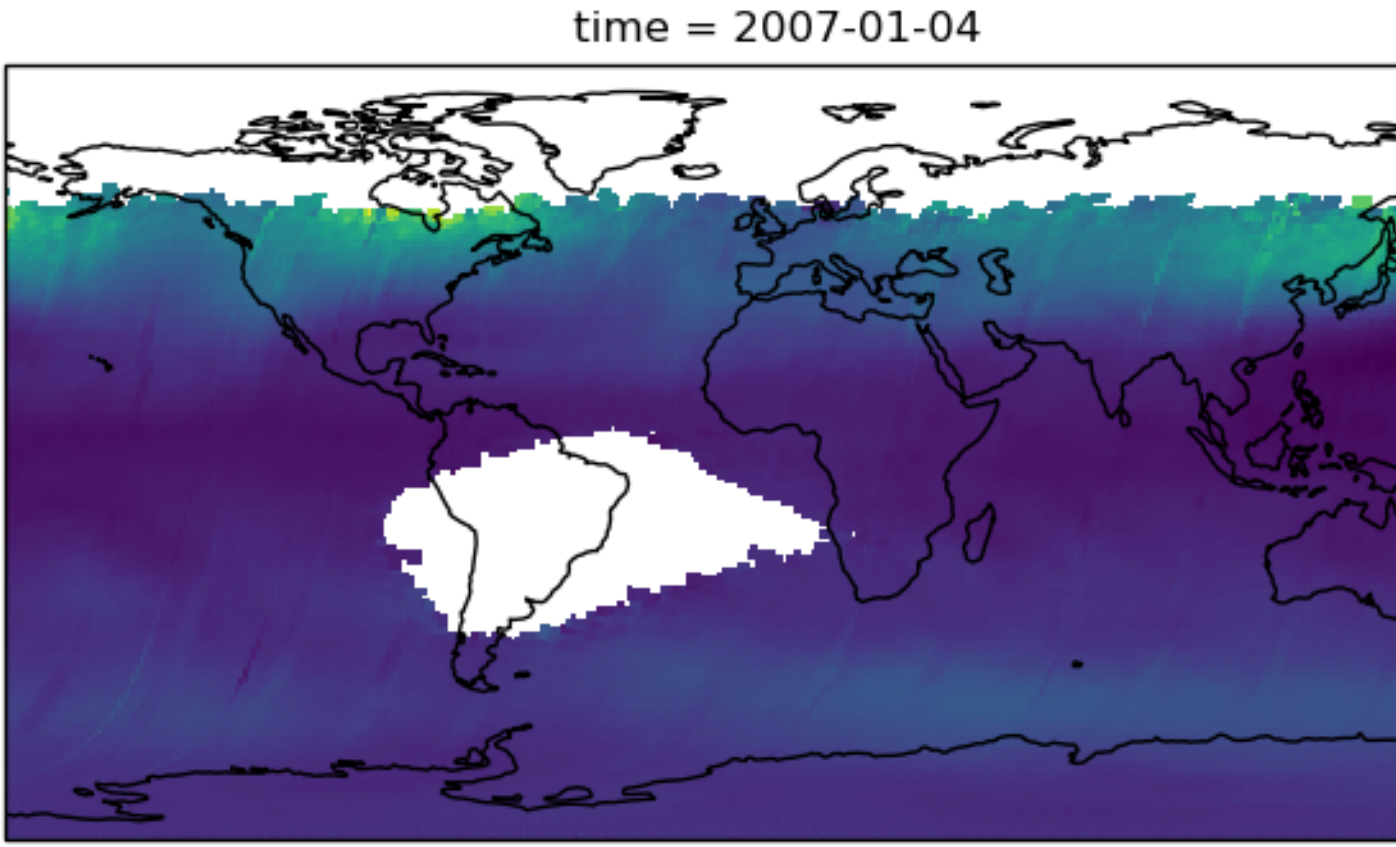

```
$ cate ws run plot_map ds=@oz_tot var=O3_du_tot file=fig2.png
Running operation 'plot_map': Executing 4 workflow step(s)
Operation 'plot_map' executed.
```

3.1.3 Co-Register the Datasets

The datasets now have different lat/lon definitions. This can be verified by using `cate res print`

```
$ cate res print cfc
<xarray.Dataset>
Dimensions:       (hist_cot: 7, hist_cot_bin: 6, hist_ctp: 8, hist_ctp_bin: 7, hist_
↪phase: 2, lat: 360, lon: 720, time: 12)
Coordinates:
  * lat           (lat) float32 -89.75 -89.25 -88.75 -88.25 -87.75 -87.25 ...
  * lon           (lon) float32 -179.75 -179.25 -178.75 -178.25 -177.75 ...
```

(continues on next page)


time = 2007-01-04

(continued from previous page)

```
  * hist_cot      (hist_cot) float32 0.3 1.3 3.6 9.4 23.0 60.0 100.0
  * hist_cot_bin  (hist_cot_bin) float32 1.0 2.0 3.0 4.0 5.0 6.0
  * hist_ctp      (hist_ctp) float32 1100.0 800.0 680.0 560.0 440.0 310.0 ...
  * hist_ctp_bin  (hist_ctp_bin) float32 1.0 2.0 3.0 4.0 5.0 6.0 7.0
  * hist_phase    (hist_phase) int32 0 1
  * time          (time) float64 2.454e+06 2.454e+06 2.454e+06 2.454e+06 ...
Data variables:
    cfc           (time, lat, lon) float64 0.1076 0.3423 0.2857 0.2318 ...
```

```
$ cate res print oz_tot
<xarray.Dataset>
Dimensions:       (air_pressure: 17, lat: 180, layers: 16, lon: 360, time: 12)
Coordinates:
  * lon           (lon) float32 -179.5 -178.5 -177.5 -176.5 -175.5 -174.5 ...
  * lat           (lat) float32 -89.5 -88.5 -87.5 -86.5 -85.5 -84.5 -83.5 ...
  * layers        (layers) int32 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  * air_pressure  (air_pressure) float32 1013.0 446.05 196.35 113.63 65.75 ...
  * time          (time) datetime64[ns] 2007-01-04 2007-02-01 2007-03-01 ...
Data variables:
    O3_du_tot     (time, lat, lon) float32 260.176 264.998 267.394 265.048 ...
```

```
$ cate op list --tag geom
2 operations found
   1: coregister
   2: subset_spatial
```

will list all commands that have a tag that matches '*geom*'. To find out more about a particular operation, use `cate op info`

```
$ cate op info coregister

Operation cate.ops.coregistration.coregister
============================================

Perform coregistration of two datasets by resampling the slave dataset unto the
grid of the master. If upsampling has to be performed, this is achieved using
interpolation, if downsampling has to be performed, the pixels of the slave dataset
are aggregated to form a coarser grid.

The returned dataset will contain the lat/lon intersection of provided
master and slave datasets, resampled unto the master grid frequency.

This operation works on datasets whose spatial dimensions are defined on
pixel-registered and equidistant in lat/lon coordinates grids. E.g., data points
define the middle of a pixel and pixels have the same size across the dataset.

This operation will resample all variables in a dataset, as the lat/lon grid is
defined per dataset. It works only if all variables in the dataset have lat
and lon as dimensions.

For an overview of downsampling/upsampling methods used in this operation, please
see https://github.com/CAB-LAB/gridtools

Whether upsampling or downsampling has to be performed is determined automatically
based on the relationship of the grids of the provided datasets.
```

(continues on next page)

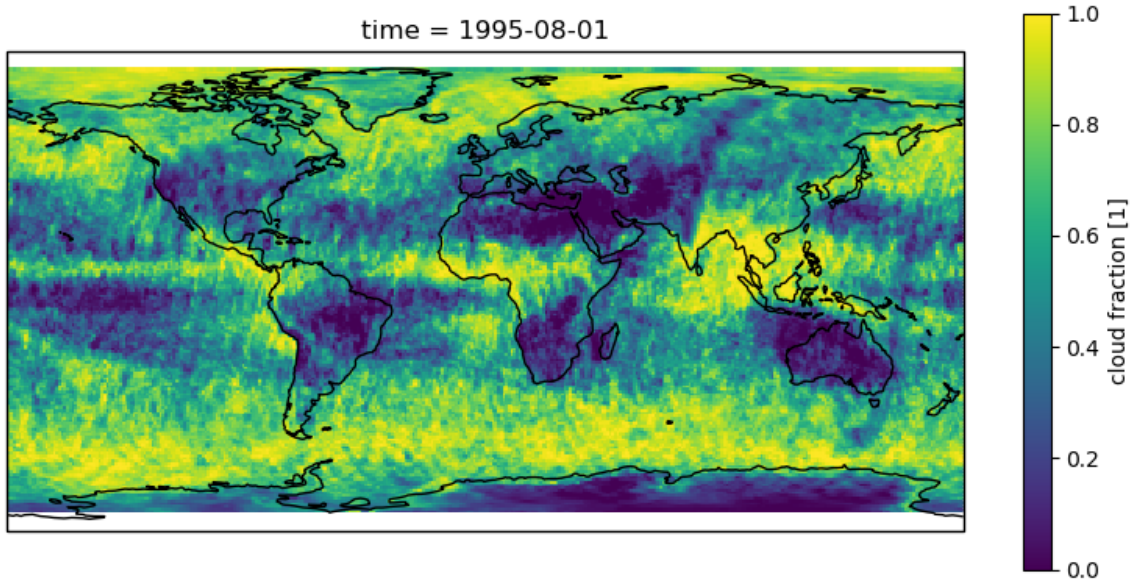
(continued from previous page)

```
Version: 1.1

Inputs:
  ds_master (Dataset)
      The dataset whose grid is used for resampling
  ds_slave (Dataset)
      The dataset that will be resampled
  method_us (str)
      Interpolation method to use for upsampling.
      default value: linear
      value set: ['nearest', 'linear']
  method_ds (str)
      Interpolation method to use for downsampling.
      default value: mean
      value set: ['first', 'last', 'mean', 'mode', 'var', 'std']

Output:
  return (Dataset)
      The slave dataset resampled on the grid of the master
      add history: True
```

To carry out coregistration, use `cate res set` again with appropriate operation parameters

```
$ cate res set cfc_res coregister ds_master=@oz_tot ds_slave=@cfc
Executing 5 workflow step(s): done
Resource "cfc_res" set.
```

```
$ cate ws run plot_map ds=@cfc_res var=cfc file=fig3.png
Running operation 'plot_map': Executing 5 workflow step(s)
Operation 'plot_map' executed.
```



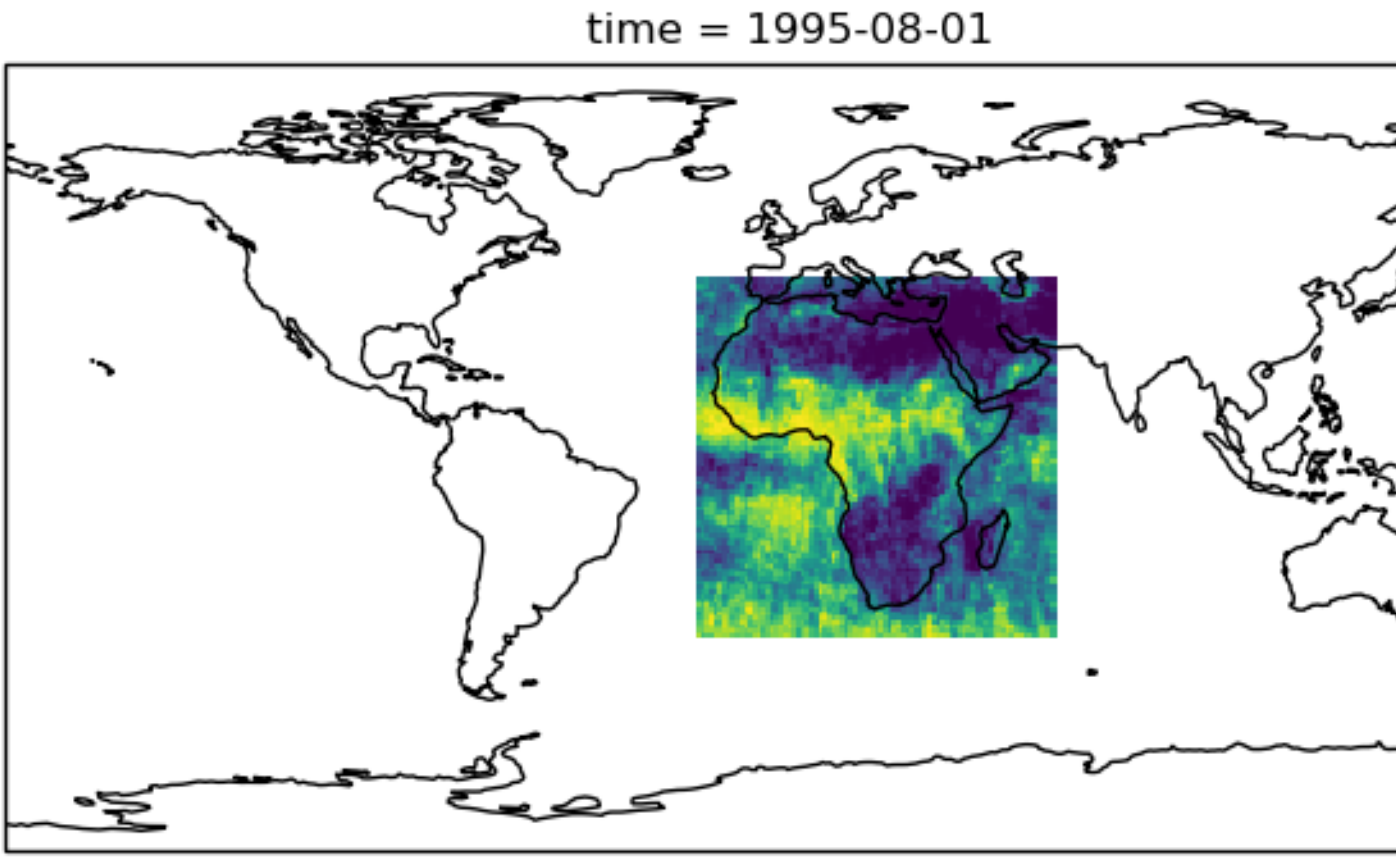
3.1.4 Spatial Filtering

To filter the datasets to contain only a particular region use the `subset_spatial` operation.

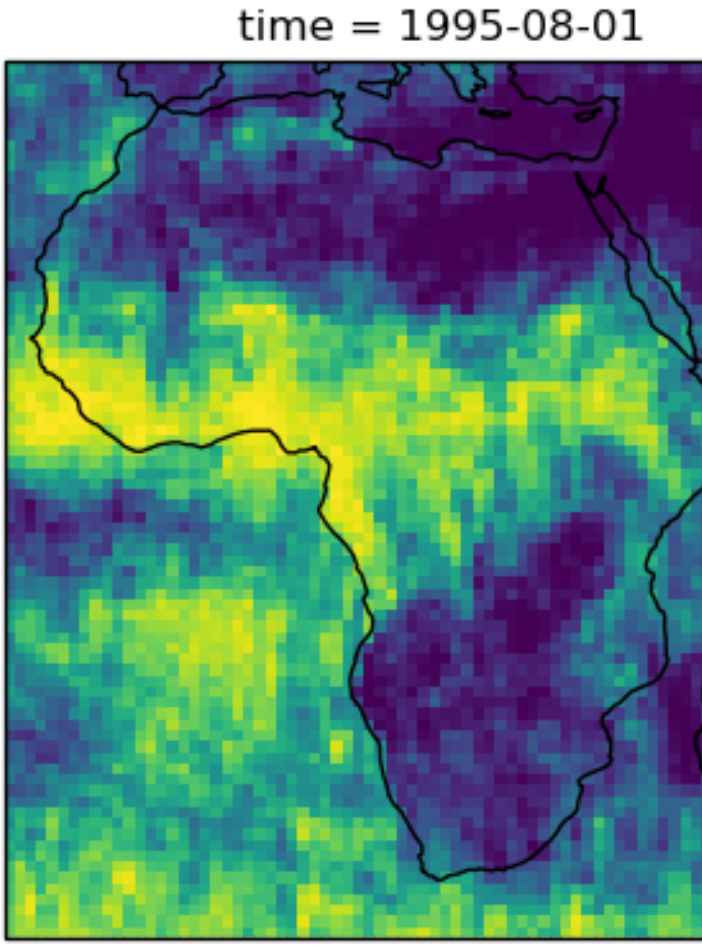
```
$ cate res set oz_africa subset_spatial ds=@oz_tot region=-20,-40,60,40
Executing 3 workflow step(s): done
Resource "oz_africa" set.
```

```
$ cate res set cc_africa subset_spatial ds=@cfc_res region=-20,-40,60,40
Executing 6 workflow step(s): done
Resource "cc_africa" set.
```

```
$ cate ws run plot_map ds=@cc_africa var=cfc file=fig4.png
Running operation 'plot_map': Executing 7 workflow step(s)
Operation 'plot_map' executed.
```




```
$ cate ws run plot_map ds=@cc_africa var=cfc region=-20,-40,60,40 file=fig5.png
Running operation 'plot_map': Executing 7 workflow step(s)
Operation 'plot_map' executed.
```


time = 1995-08-01

3.1.5 Temporal Filtering

To further filter the datasets to contain only a particular time range, use `subset_temporal` operation

```
$ cate res set oz_africa_janoct subset_temporal ds=@oz_africa time_range=2007-01-01,
↪2007-10-30
$ cate res set cc_africa_janoct subset_temporal ds=@cc_africa time_range=2007-01-01,
↪2007-10-30
```

3.1.6 Extract Time Series

We'll extract spatial mean timeseries from both datasets using `tseries_mean` operation.

```
$ cate res set cc_africa_ts tseries_mean ds=@cc_africa_janoct var=cfc
Executing 8 workflow step(s): done
Resource "cc_africa_ts" set.
```

```
$ cate res set oz_africa_ts tseries_mean ds=@oz_africa_janoct var=O3_du_tot
Executing 5 workflow step(s): done
Resource "oz_africa_ts" set.
```

This creates datasets that contain mean and std variables for both time-series.

3.1.7 Time Series Plot

To plot the time-series and save the `plot` operation can be used together with `cate ws run` operation:

```
$ cate ws run plot ds=@cc_africa_ts var=cfc file=fig6.png
Running operation 'plot': Executing 11 workflow step(s)
Operation 'plot' executed.
```

```
$ cate ws run plot ds=@oz_africa_ts var=O3_du_tot file=fig7.png
Running operation 'plot': Executing 11 workflow step(s)
Operation 'plot' executed.
```

3.1.8 Product-Moment Correlation

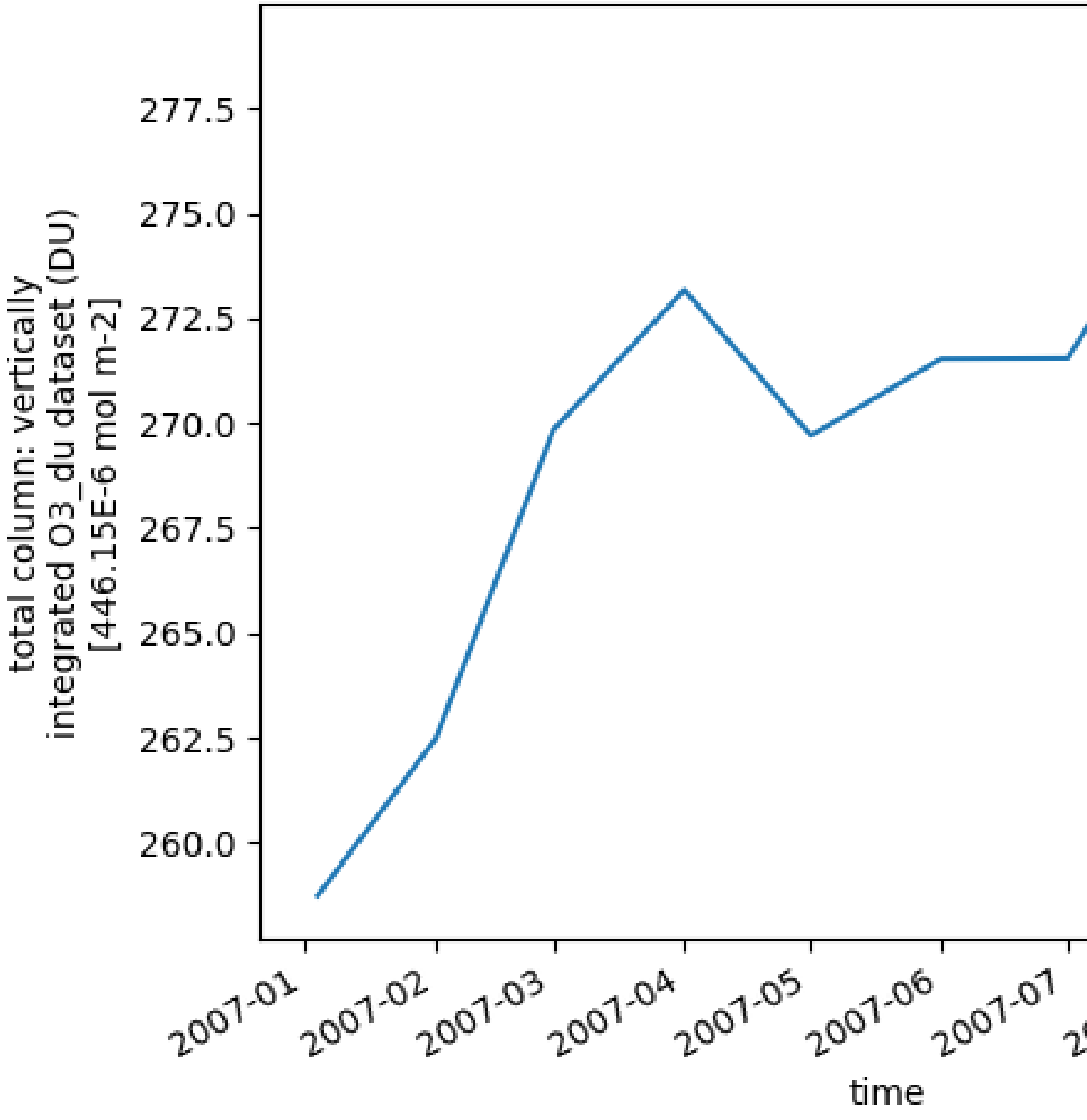
To carry out a product-moment correlation on the mean time-series, the `pearson_correlation_scalar` operation can be used.

```
$ cate op list --tag correlation
2 operations found
  0: pearson_correlation
  1: pearson_correlation_scalar
```

```
$ cate res set pearson pearson_correlation ds_y=@cc_africa_ts ds_x=@oz_africa_ts var_
↪y=cfc var_x=O3_du_tot
Executing 12 workflow step(s): done
Resource "pearson" set.
```

This will calculate the correlation coefficient along with the associated p_value for both mean time-series. We can view the result using `cate res print`, or save it using `cate res write`:


0.47
0.46
0.45
0.44
0.43
0.42
0.41
cloud fraction [1]
2007-01
2007-02
2007-03
2007-04
2007-05
2007-06
2007-07
Time in days


total column: vertically integrated O3_du dataset (DU) [446.15E-6 mol m-2]
277.5
275.0
272.5
270.0
267.5
265.0
262.5
260.0
2007-01
2007-02
2007-03
2007-04
2007-05
2007-06
2007-07
time

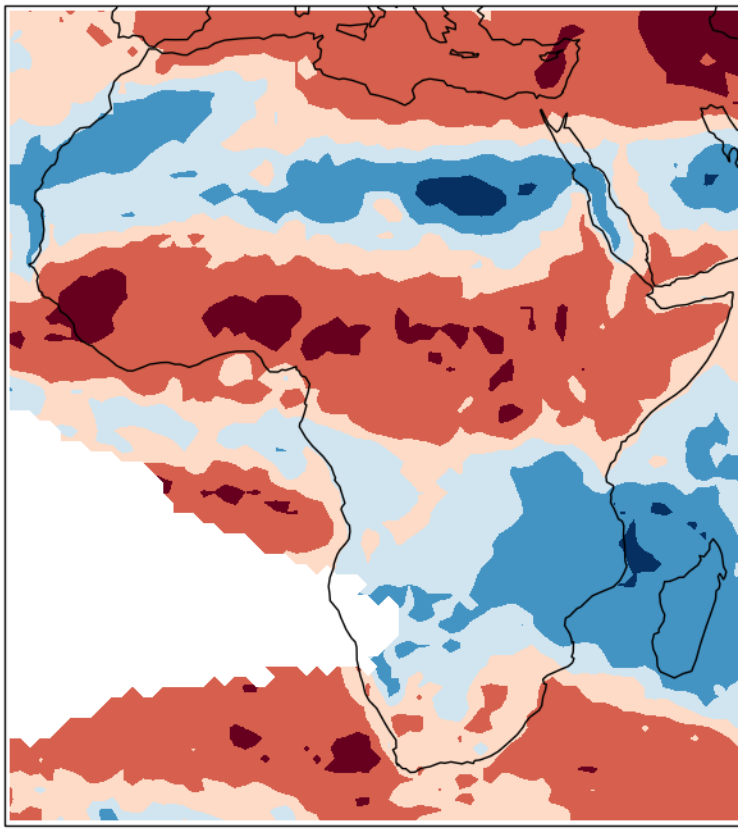
```
$ cate res print pearson
{'corr_coef': -0.2924, 'p_value': 0.4123}
```

```
$ cate res write pearson pearson.txt
```

To carry out a pixel by pixel correlation of two coregistered time/lat/lon datasets such that the result is a map of correlation coefficients or the corresponding probability values, one can use `pearson_correlation`:

```
$ cate res set pearson_map pearson_correlation ds_y=@cc_africa_janoct ds_x=@oz_africa_
↪janoct var_y=cfc var_x=O3_du_tot
Executing 10 workflow step(s): done
Resource "pearson_map" set.
```

```
$ cate ws run plot_map ds=@pearson_map var=corr_coef lat_min=-40 lat_max=40 lon_min=-
↪20 lon_max=60 file=fig8.png
Running operation 'plot_map': Executing 13 workflow step(s)
Operation 'plot_map' executed.
```



3.1.9 More Examples

More CLI Examples can be found in the scripts directory within Cate's GitHub repository.

3.2 Using the API

A demonstration of how to apply the Cate's Python API to the use case described here is given in the cate-uc09.ipynb notebook example.

CHAPTER 4

User Manual

4.1 Setup

Cate at its core is a Python package that provides Cate's command-line interface (CLI) and application programming interface (API). In addition, the Python package provides a visualisation and processing service for *Cate Desktop*, Cate's graphical user interface (GUI).

For *Cate Desktop*, we provide an installer for the Windows, Mac OS X, and Linux operating systems. The Cate Desktop installer will also ensure the Cate Python package is installed. If it can't find an existing or compatible Cate Python package, it will install a new or update an existing one.

If you only want the Cate CLI or API, you can install just the Python package into a dedicated Miniconda or Anaconda Python 3 environment. In this case, please read *Installing Cate (CLI, API)*.

4.1.1 System Requirements

Hardware: It is recommended to use an up-to-date computer, with at least 8GB of RAM and a multi-core CPU. The most important bottlenecks will first be the data transfer rate from local data caches into the executing program, so it is advised to use fast solid state disks. Secondly, the internet connection speed matters, because Cate will frequently have to download data from remote services in order to cache it locally.

Operating Systems: Cate is supposed to work on up-to-date Windows, Mac OS X, and Linux operating systems.

4.1.2 Installing Cate Desktop (GUI)

First time installs

The Cate Desktop installer for your platform is available from the Cate website.

Should the website be unavailable, you can get the installer directly from the releases page in Cate Desktop's GitHub repository:

- `cate-desktop-2.x.y.dmg` and for OS X;

- `cate-desktop-2.x.y-x86_64.AppImage` for Linux;
- `cate-desktop-setup-2.x.y.exe` for Windows.

All Cate Desktop installers are quite light-weight and executed by double clicking them.

They don't require any extra user input up to the point where no existing or compatible Cate Python package is found. In this case, Cate's *Setup process* is run:

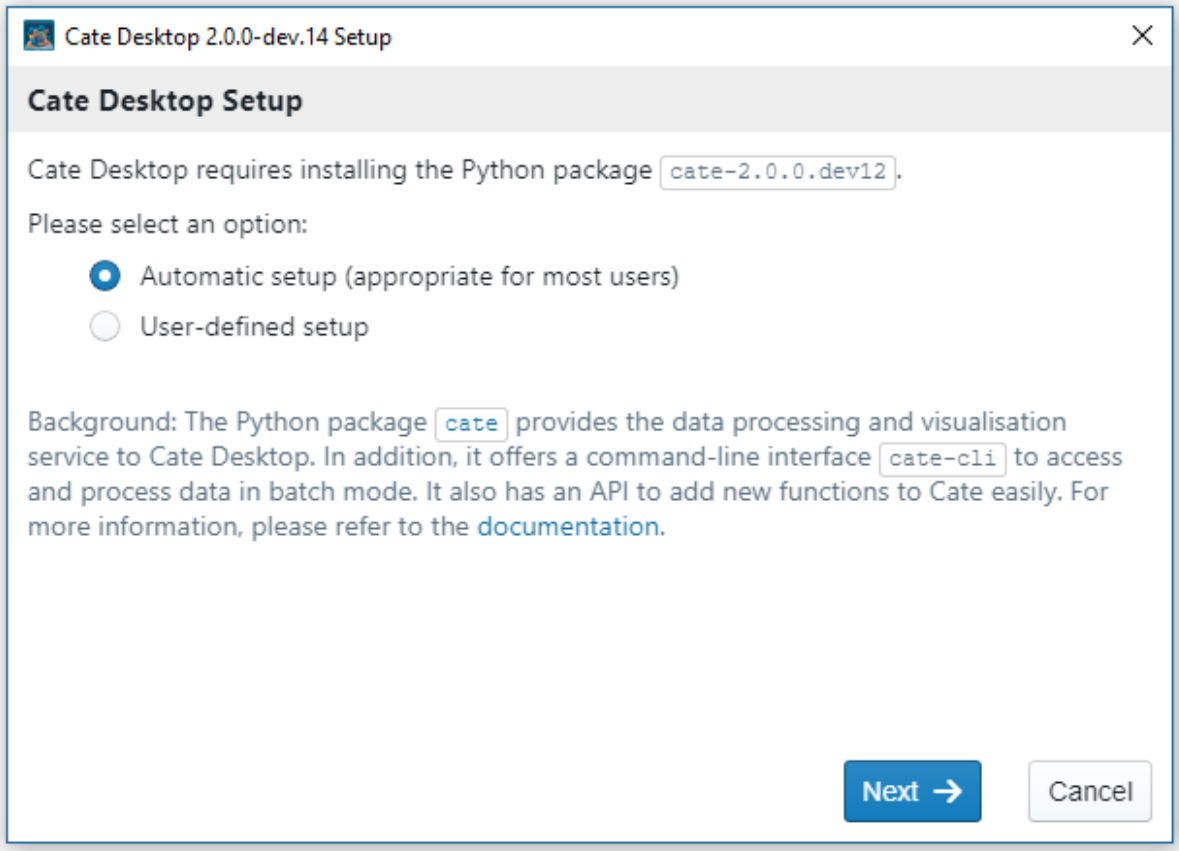


Fig. 4.1: Setup dialog - start screen

You can just click **Next** button to use *Automatic setup* with default settings. To see what these settings are, you could select **User-defined setup** and press **Next** in which case the default settings are shown:

Pressing **Next** will perform the following setup steps for a new Cate Python package:

1. Downloading a Miniconda installer;
2. Running the Miniconda in installer in the background to install a dedicated Python environment;
3. Installing the Python conda package `cate-cli` into that environment.

For an existing, outdated Cate Python package it will just update it to the required version and also update all required 3rd-party Python packages.

After successful installation, press **End** to start Cate Desktop:

Should you encounter any problems with the setup, please consider filing an error report in the Cate issue tracker.

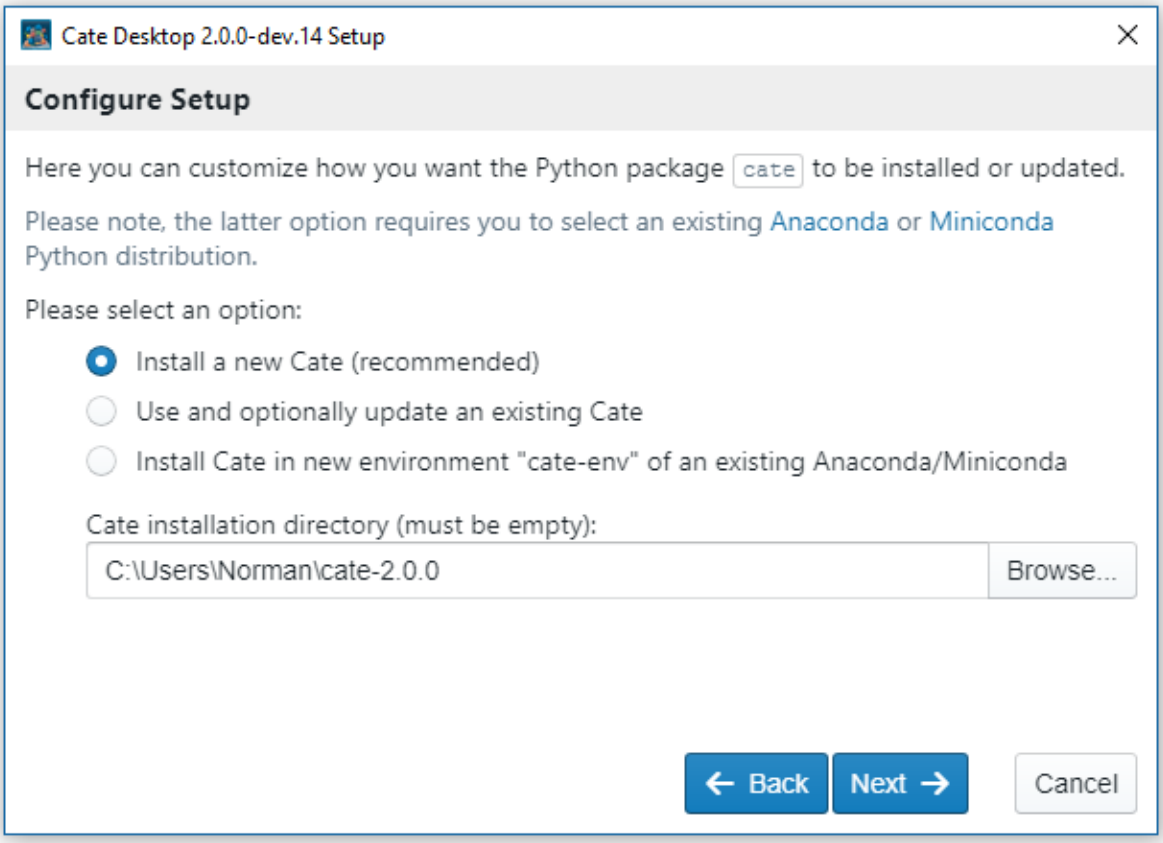


Fig. 4.2: Setup dialog - user defined settings

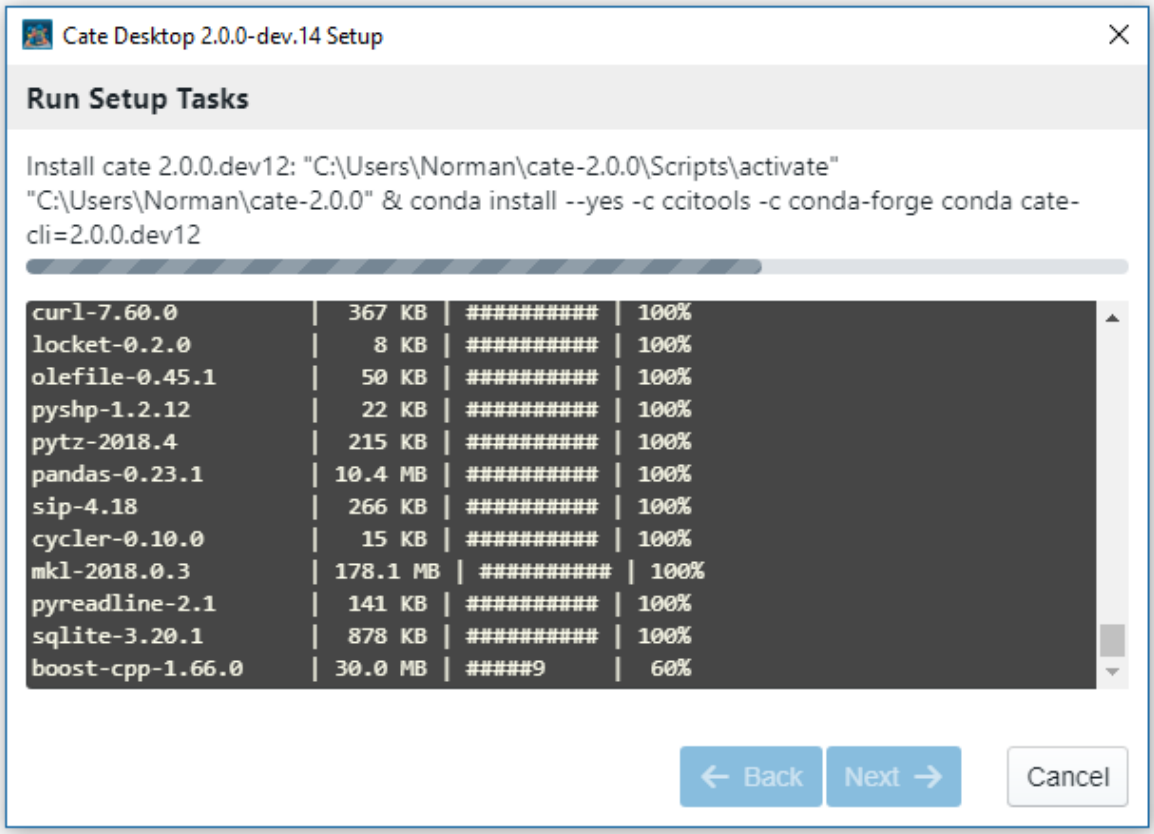


Fig. 4.3: Setup dialog - setup in progress

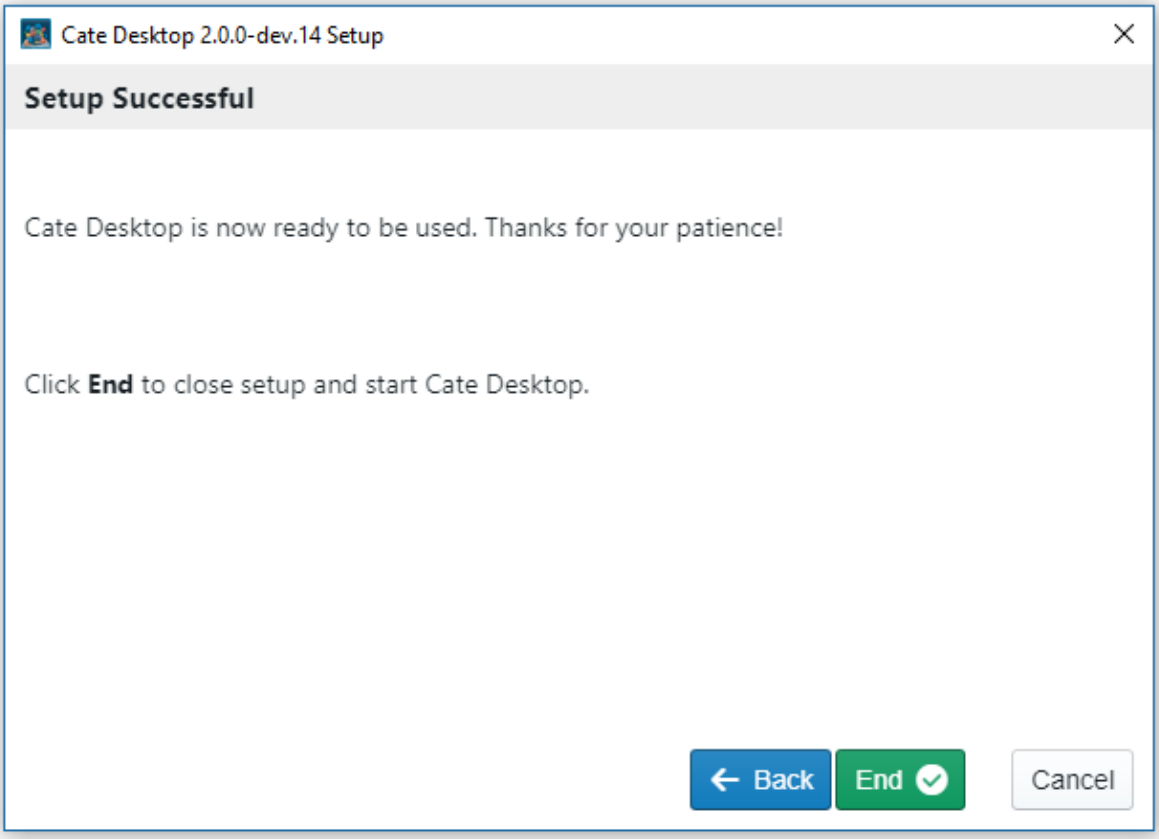


Fig. 4.4: Setup dialog - setup successful

Updating

By default, Cate Desktop is supposed to keep itself up-to-date automatically. Once the update is installed, Cate Desktop might detect an outdated Cate Python package. In this case the *Setup process* described above is run again to update the Python package to the required version.

In case the update procedure fails, uninstall Cate Desktop, then download the latest version for your operating system and install again.

The auto-update feature of Cate Desktop can be disabled in the **Preferences**:

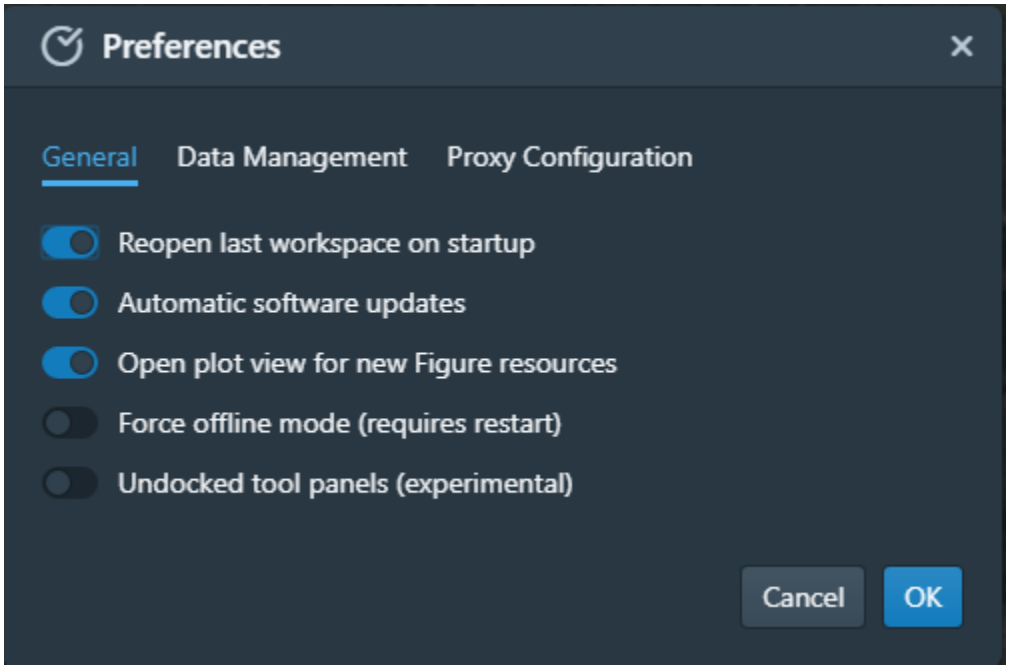


Fig. 4.5: Preferences Dialog / General

4.1.3 Installing Cate (CLI, API)

First time installs

The Cate Python package requires a *Conda environment* for Python 3.6+ either provided by a Miniconda or Anaconda installation.

If you haven't yet installed either of the two, we recommend you install Miniconda for Python 3 first.

With Miniconda/Anaconda installed and accessible (installation path should be on `PATH` environment variable) open a shell / terminal window (Windows users type "cmd" in search field of start menu).

The steps are:

1. create a dedicate Python environment for Cate so it doesn't interfere with other Python packages you might already have installed;
2. activate that newly create Python environment for Cate;
3. install the Cate Python package;
4. test the installation by invoking the Cate command-line interface.

Mac OS / Linux:

```
$ conda env create -n cate-env
$ source activate cate-env
$ conda install -c conda-forge -c ccitools cate-cli
$ cate --help
```

Windows:

```
> conda env create -n cate-env
> activate cate-env
> conda install -c conda-forge -c ccitools cate-cli
> cate --help
```

Updating

You can easily update an existing Cate installation using the `cate upd` command:

```
$ cate upd
```

Or you use Conda to install the latest version:

```
$ conda update -c conda-forge -c ccitools cate-cli
```

4.1.4 Installing from Sources

If you are a developer you may wish to build and install Cate from Python sources. In this case, please follow the instructions given in the Cate README on GitHub.

After building and installing the Cate Python package from sources you can build an run Cate Desktop from sources by following the instructions given in the Cate Desktop README on GitHub.

4.2 Configuration

4.2.1 Configuration file

Cate's configuration settings are read from `.cate/conf.py` located in the current user's home directory.

Given here is an overview of the possible configuration settings:

`data_stores_path` Directory where Cate stores information about data stores and also saves local data files synchronized with their remote versions. Use the tilde '~' (also on Windows) within the path to point to your home directory. This directory can become rather populated once after a while and it is advisable to place it where there exists a high transfer rate and sufficient capacity. Ideally, you would let it point to a dedicated solid state disc (SSD). The default value for `data_stores_path` is the `~/.cate/data_stores` directory.

`dataset_persistence_format` Names the data format to be used when persisting datasets in the workspace. Possible values are 'netcdf4' or 'zarr' (much faster, but still experimental).

`use_workspace_imagery_cache` If set to `True`, Cate will maintain a per-workspace cache for imagery generated from dataset variables. Such cache can accelerate image display, however at the cost of disk space.

default_res_pattern Default prefix for names generated for new workspace resources originating from opening data sources or executing workflow steps. This prefix is used only if no specific prefix is defined for a given operation.

included_ds_ids If `included_ds_ids` is a list, its entries are expected to be wildcard patterns for the identifiers of data sources to be included. By default, or if `included_ds_ids` is None, all data sources are included.

excluded_ds_ids If `excluded_ds_ids` is a list, its entries are expected to be wildcard patterns for the identifiers of data sources to be excluded. By default, or if `excluded_ds_ids` is None, no data sources are excluded. If both `included_ds_ids` and `excluded_ds_ids` are lists, we first include data sources using `included_ds_ids` then remove entries that match any result from applying `excluded_data_sources`.

default_variables Configure names of variables that will be initially selected once a new dataset resource is opened in the GUI. Its value must be a set (`{...}`) of variable names.

variable_display_settings Configure / overwrite default variable display settings as used in various plot_<type>() operations and in the Cate Desktop GUI. Each entry maps a variable name to a dictionary with the following entries: * `color_map` - name of a color map taken from from Matplotlib Color Maps Reference * `display_min` - minimum variable value that corresponds to the lower end of the color map * `display_max` - maximum variable value that corresponds to the upper end of the color map

For example::

```
variable_display_settings = {
    'my_var': dict(color_map='viridis', display_min=0.1, display_max=0.
↪8),
}
```

default_color_map Default color map to be used for any variable not configured in 'variable_display_settings' 'default_color_map' must be the name of a color map taken from from Matplotlib Color Maps Reference. If not specified, the ultimate default is `'inferno'`.

4.2.2 Environment variables

CATE_ESA_CCI_ODP_DATA_STORE_PATH Overrides the location of the ESA CCI ODP data store directory whose parent directory would otherwise be given by the `data_stores_path` configuration setting.

CATE_LOCAL_DATA_STORE_PATH Overrides the location of the local data store directory whose parent directory would otherwise be given by the `data_stores_path` configuration setting.

HTTP_PROXY, http_proxy, HTTPS_PROXY, https_proxy, SOCKS_PROXY, socks_proxy Recognized proxy server hosts.

NO_PROXY, no_proxy Comma-separated lists of hosts that should bypass the proxy server.

4.3 Command-Line Interface

4.3.1 Overview

The CCI Toolbox comprises a single command-line executable, which is called `cate` and is available after installing the CCI Toolbox on your computer. See section *Setup* for more information. The command-line interface allows for accessing local and remote datasets as well as running virtually all CCI Toolbox operations on them.

The most easy way to use `cate` is running the `cate-cli` script found in `bin` directory of your CCI Toolbox installation directory. Windows and Unix users will find a link to this script in their start menu or on their desktop. Opening the link will open a new console / terminal window configured to run `cate`.

Developers only: If you build and install the CCI Toolbox from Python sources into your current Python environment, `cate` will be registered as an executable script. It can be found as `$PYTHON_PREFIX/bin/cate.sh` on Unix systems and as `%PYTHON_PREFIX%\\Scripts\\cate.exe` on Windows systems where `PYTHON_PREFIX` is the path to the current Python environment.

In the console / terminal window type:

```
cate -h
```

This should output the following usage help::

```
usage: cate [-h] [--version] [--traceback] [--license] [--docs] COMMAND ...

ESA CCI Toolbox (Cate) command-line interface, version 2.0

positional arguments:
  COMMAND      One of the following commands. Type "COMMAND -h" to get
               command-specific help.
    ds         Manage data sources.
    op         Manage data operations.
    ws         Manage workspaces.
    res        Manage workspace resources.
    run        Run an operation or Workflow file.
    io         Manage supported data and file formats.
    upd        Update an existing cate environment to a specific or to the
               latest cate version

optional arguments:
  -h, --help   show this help message and exit
  --version    show program's version number and exit
  --traceback  show (Python) stack traceback for the last error
  --license    show software license and exit
  --docs       show software documentation in a browser window
```

`cate` uses up to two sub-command levels. Each sub-command has its own set of options and arguments and can display help when used with the option `` `--help `` or `-h`. The first sub-command level comprises the following list of commands:

- *cate ds - Dataset Management*
- *cate op - Operation Management*
- *cate run - Running Operations and Workflows*

The following first level sub-commands are used to work interactively with datasets and operations:

- *cate ws: Workspace Management*
- *cate res - Workspace Resources Management*

When you encounter any error while using `cate` and you want to report the problem to the development team, we kindly ask you to rerun the command with option `--traceback` and include the Python stack traceback with a short description of your problem.

4.3.2 Examples

The following examples shall help you understand the basic concepts behind the various `cate` commands.

Manage datasets

To list all available data sources, type:

```
cate ds list
```

To query all data sources that have `ozone` in their name, type:

```
cate ds list -n ozone
```

To get more detailed information on a specific data source, e.g. `esacci.OZONE.mon.L3...`, type:

```
cate ds info esacci.OZONE.mon.L3.NP.multi-sensor.multi-platform.MERGED.fv0002.r1
```

To add a local data source from all NetCDF files in e.g. `data/sst_v3` and name it e.g. `SSTV3`, type:

```
cate ds def SSTV3 data/sst_v3/*.nc
```

Make sure it is there:

```
cate ds list -n SSTV3
```

To make a temporal subset ECV data source locally available, i.e. avoid remote data access during its usage:

```
cate ds copy esacci.OZONE.mon.L3.NP.multi-sensor.multi-platform.MERGED.fv0002.r1 -t␣
↪2006-01-01,2007-12-31
```

The section Configuration in *Setup* describes, how to configure the location of directory in which Cate stores such synchronised data.

Inspect available operations

To list all available operations, type:

```
cate op list
```

To display more details about a particular operation, e.g. `tseries_point`, type:

```
cate op info tseries_point
```

Run an operation

The `cate run` command is used to execute single operations. The `open` and `read` options are used to ingest datasets which can then be referenced by name. A `write` option allows to write the operation result into a file.

To run the `tseries_point` operation on a dataset, e.g. the `local.SSTV3` (from above), at lat=0 and lon=0, type:

```
cate run --open ds=local.SSTV3 --write ts2.nc tseries_point ds=ds lat=0 lon=0
```

To run the `tseries_point` operation on a netCDF file, e.g. `test/ui/precip_and_temp.nc` at lat=0 and lon=0, type:

```
cate run --read ds=test/ui/precip_and_temp.nc --write ts2.nc tseries_point ds=ds␣
↪lat=0 lon=0
```

Interactive session

The following command sequence is a simple example for an interactive session using the Cate command-line:

```
cate ws new
cate res open sst local.SSTV3
cate res set sst_ts tseries_point ds=@sst lat=0 lon=0
cate res plot sst_ts
cate res write sst_ts sst_ts.nc
cate ws status
```

The steps above explained:

1. `cate ws new` is used to create a new in-memory *workspace*. A workspace can hold any number of named *workspace resources* which may refer to opened datasets or any other ingested or computed objects.
2. `cate res open` is used to open a dataset from the available data stores and assign the opened dataset to the workspace resource `sst`. Accordingly, `cate res read` could have been used to read from a local netCDF file.
3. `cate res set` assigns the result of the `tseries_point` operation to workspace resource `sst_ts`. Note the at-character "@" used as prefix for the input `ds`. This indicates that value for input `ds` of step `tseries_point` will be retrieved "at" the `open` step named `sst`. It establishes a connection between step `open` and `tseries_point`. In fact, this is the way processing graphs are constructed using the CLI.
4. `cate res plot` plots the workspace resource `sst_ts`.
5. `cate res write` writes the workspace resource `sst_ts` to a netCDF file `./sst_ts.nc`.
6. `cate ws status` shows the current workspace status and lists all workspace resource assignments.

We could now save the current workspace state and close it:

```
cate ws save
cate ws close
```

`cate ws save` creates a hidden sub-directory `.cate-workspace` and herewith makes the current directory a *workspace directory*. `cate` uses this hidden directory to persist the workspace state information. At a later point in time, you could `cd` into any of your workspace directories, and:

```
cate ws open
cate ws status
```

in order to reopen it, display its status, and continue interactively working with its resources.

The following subsections provide detailed information about the `cate` commands.

4.3.3 `cate ds` - Dataset Management

Provides a set of sub-commands used to manage climate data sources. Data sources are used to open local and remote datasets which are input to various analysis and processing operations. Type "cate op -h" to find out more about available operations.

```
usage: cate ds [-h] COMMAND ...
```

Positional Arguments

COMMAND Possible choices: list, info, add, del, copy

One of the following commands. Type "COMMAND -h" for help.

Sub-commands:

list

List all available data sources

```
cate ds list [-h] [--name NAME] [--coverage]
```

Named Arguments

--name, -n List only data sources named NAME or that have NAME in their name. The comparison is case insensitive.

--coverage, -c Also display temporal coverage

Default: False

info

Display information about a data source.

```
cate ds info [-h] [--var] [--local] DS
```

Positional Arguments

DS A data source name. Type "cate ds list" to show all possible data source names.

Named Arguments

--var, -v Also display information about contained dataset variables.

Default: False

--local, -l Also display temporal coverage of cached datasets.

Default: False

add

Add a new local data source using a file pattern.

```
cate ds add [-h] DS FILE [FILE ...]
```

Positional Arguments

DS A name for the data source.

FILE A list of files comprising this data source. The files can contain the wildcard characters "*" and "?".

del

Removes a data source from local data store.

```
cate ds del [-h] [-k] [-y] DS
```

Positional Arguments

DS A name for the data source.

Named Arguments

-k, --keep_files Do not ask for confirmation.

Default: False

-y, --yes Do not ask for confirmation.

Default: False

copy

Makes a local copy of any other data source. The copy may be limited to a subset by optional constraints.

```
cate ds copy [-h] [--name NAME] [--time TIME] [--region REG] [--vars VARS]
             REF_DS
```

Positional Arguments

REF_DS A name of origin data source.

Named Arguments

--name, -n	A name for new data source.
--time, -t	Time range constraint. Use format "YYYY-MM-DD,YYYY-MM-DD".
--region, -r	Region constraint. Use format: "min_lon,min_lat,max_lon,max_lat".
--vars, -v	Names of variables to be included. Use format "pattern1,pattern2,..."

4.3.4 `cate op` - Operation Management

Provides a set of commands to inquire the available operations used to analyse and process climate datasets.

```
usage: cate op [-h] COMMAND ...
```

Positional Arguments

COMMAND	Possible choices: list, info
	One of the following commands. Type "COMMAND -h" for help.

Sub-commands:

list

List operations.

```
cate op list [-h] [--name NAME] [--tag TAG] [--deprecated] [--internal]
```

Named Arguments

--name, -n	List only operations with name NAME or that have NAME in their name. The comparison is case insensitive.
--tag, -t	List only operations tagged by TAG or that have TAG in one of their tags. The comparison is case insensitive.
--deprecated, -d	List deprecated operations.
	Default: False
--internal, -i	List operations tagged "internal".
	Default: False

info

Show usage information about an operation.

```
cate op info [-h] OP
```

Positional Arguments

OP Fully qualified operation name.

4.3.5 `cate run` - Running Operations and Workflows

Runs the given operation or Workflow file with the specified operation arguments. Argument values may be constant values or the names of data loaded by the –open or –read options. Type "cate op list" to list all available operations. Type "cate op info" to find outwhich arguments are supported by a given operation.

```
usage: cate run [-h] [-m] [-o DS_EXPR] [-r FILE_EXPR] [-w FILE_EXPR] OP ...
```

Positional Arguments

OP Fully qualified operation name or Workflow file. Type "cate op list" to list available operators.

... Operation arguments given as KEY=VALUE. KEY is any supported input by OP. VALUE depends on the expected data type of an OP input. It can be a True, False, a string, a numeric constant, one of the names specified by the –open and –read options, or a Python expression. Type "cate op info OP" to print information about the supported OP input names to be used as KEY and their data types to be used as VALUE.

Named Arguments

-m, --monitor Display progress information during execution.

Default: False

-o, --open Open a dataset from DS_EXPR. The DS_EXPR syntax is NAME=DS[,START[,END]]. DS must be a valid data source name. Type "cate ds list" to show all known data source names. START and END are dates and may be used to create temporal data subsets. The dataset loaded will be assigned to the arbitrary name NAME which is used to pass the datasets or its variablesas an OP argument. To pass a variable use syntax NAME.VAR_NAME.

-r, --read Read object from FILE_EXPR. The FILE_EXPR syntax is NAME=PATH[,FORMAT]. Type "cate io list -r" to see which formats are supported.If FORMAT is not provided, file format is derived from the PATH's filename extensions or file content. NAME may be passed as an OP argument that receives a dataset, dataset variable or any other data type. To pass a variable of a dataset use syntax NAME.VAR_NAME

-w, --write Write result to FILE_EXPR. The FILE_EXPR syntax is [NAME=]PATH[,FORMAT]. Type "cate io list -w" to see which formats are supported.If FORMAT is not provided, file format is derived from the object type and the PATH's filename extensions. If OP returns multiple named output values, NAME is used to identify them. Multiple -w options may be used in this case.

4.3.6 `cate ws`: Workspace Management

Used to create, open, save, modify, and delete workspaces. Workspaces contain named workflow resources, which can be datasets read from data stores, or any other data objects originating from applying operations to datasets and other data objects. The origin of every resource is stored in the workspace's workflow description. Type "cate res -h" for more information about workspace resource commands.

```
usage: cate ws [-h] COMMAND ...
```

Positional Arguments

COMMAND Possible choices: init, new, open, close, save, run, del, clean, status, list, exit

One of the following commands. Type "COMMAND -h" for help.

Sub-commands:

init

Initialize workspace.

```
cate ws init [-h] [-d DIR] [--desc DESCRIPTION]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

--desc Workspace description.

new

Create new in-memory workspace.

```
cate ws new [-h] [-d DIR] [--desc DESCRIPTION]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

--desc Workspace description.

open

Open workspace.

```
cate ws open [-h] [-d DIR]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

close

Close workspace.

```
cate ws close [-h] [-d DIR] [-a] [-s]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

-a, --all Close all workspaces. Ignores DIR option.

Default: False

-s, --save Save modified workspace before closing.

Default: False

save

Save workspace.

```
cate ws save [-h] [-d DIR] [-a]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

-a, --all Save all workspaces. Ignores DIR option.

Default: False

run

Run operation.

```
cate ws run [-h] [-d DIR] OP ...
```

Positional Arguments

OP	Operation name or Workflow file path. Type "cate op list" to list available operations.
...	Operation arguments given as KEY=VALUE. KEY is any supported input by OP. VALUE depends on the expected data type of an OP input. It can be either a value or a reference an existing resource prefixed by the add character "@". The latter connects to operation steps with each other. To provide a (constant)value you can use boolean literals True and False, strings, or numeric values. Type "cate op info OP" to print information about the supported OP input names to be used as KEY and their data types to be used as VALUE.

Named Arguments

-d, --dir	The workspace's base directory. If not given, the current working directory is used. Default: "."

del

Delete workspace.

```
cate ws del [-h] [-d DIR] [-y]
```

Named Arguments

-d, --dir	The workspace's base directory. If not given, the current working directory is used. Default: "."
-y, --yes	Do not ask for confirmation. Default: False

clean

Clean workspace (removes all resources).

```
cate ws clean [-h] [-d DIR] [-y]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

-y, --yes Do not ask for confirmation.

Default: False

status

Print workspace information.

```
cate ws status [-h] [-d DIR]
```

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

list

List all opened workspaces.

```
cate ws list [-h]
```

exit

Exit interactive mode. Closes all open workspaces.

```
cate ws exit [-h] [-y] [-s]
```

Named Arguments

-y, --yes Do not ask for confirmation.

Default: False

-s, --save Save any modified workspaces before closing.

Default: False

4.3.7 `cate res` - Workspace Resources Management

Used to set, run, open, read, write, plot, etc. workspace resources. All commands expect an opened workspace. Type "cate ws -h" for more information about workspace commands.

```
usage: cate res [-h] COMMAND ...
```

Positional Arguments

COMMAND Possible choices: open, read, write, set, rename, del, print, plot

One of the following commands. Type "COMMAND -h" for help.

Sub-commands:

open

Open a dataset from a data source and set a resource.

```
cate res open [-h] [-d DIR] NAME DS [START] [END] [REGION] [VAR_NAMES]
```

Positional Arguments

NAME Name of the new target resource.

DS A data source named DS. Type "cate ds list" to list valid data source names.

START Start date. Use format "YYYY[-MM[-DD]]".

END End date. Use format "YYYY[-MM[-DD]]".

REGION Region constraint. Use format "min_lon,min_lat,max_lon,max_lat".

VAR_NAMES Names of variables to be included. Use format "pattern1,pattern2,pattern3".

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

read

Read an object from a file and set a resource.

```
cate res read [-h] [-d DIR] [-f FORMAT] NAME FILE
```

Positional Arguments

NAME	Name of the new target resource.
FILE	File path.

Named Arguments

-d, --dir	The workspace's base directory. If not given, the current working directory is used. Default: "."
-f, --format	File format. Type "cate io list -r" to see which formats are supported.

write

Write a resource to a file.

```
cate res write [-h] [-d DIR] [-f FORMAT] NAME FILE
```

Positional Arguments

NAME	Name of an existing resource.
FILE	File path.

Named Arguments

-d, --dir	The workspace's base directory. If not given, the current working directory is used. Default: "."
-f, --format	File format. Type "cate io list -w" to see which formats are supported.

set

Set a resource from the result of an operation.

```
cate res set [-h] [-d DIR] [-o] NAME OP ...
```

Positional Arguments

NAME	Name of the target resource to be set. Use -o to overwrite an existing NAME.
OP	Operation name. Type "cate op list" to list available operation names.

... Operation arguments given as KEY=VALUE. KEY is any supported input by OP. VALUE depends on the expected data type of an OP input. It can be either a value or a reference an existing resource prefixed by the add character "@". The latter connects to operation steps with each other. To provide a (constant)value you can use boolean literals True and False, strings, or numeric values. Type "cate op info OP" to print information about the supported OP input names to be used as KEY and their data types to be used as VALUE.

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

-o, --overwrite Overwrite an existing workflow step / target resource with same NAME.

Default: False

rename

Rename a resource.

```
cate res rename [-h] [-d DIR] NAME NEW_NAME
```

Positional Arguments

NAME Resource name.

NEW_NAME New resource name.

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

del

Delete a resource.

```
cate res del [-h] [-d DIR] NAME
```

Positional Arguments

NAME Resource name.

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

print

If EXPR is omitted, print value of all current resources.Otherwise, if EXPR identifies a resource, print its value.Else print the value of a (Python) expression evaluated in the context of the current workspace.

```
cate res print [-h] [-d DIR] [EXPR]
```

Positional Arguments

EXPR Name of an existing resource or a valid (Python) expression.

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

plot

Plot a resource or the value of a (Python) expression evaluated in the context of the current workspace.

```
cate res plot [-h] [-d DIR] [-v [VAR]] [-o [FILE]] EXPR
```

Positional Arguments

EXPR Name of an existing resource or any (Python) expression.

Named Arguments

-d, --dir The workspace's base directory. If not given, the current working directory is used.

Default: "."

-v, --var Name of a variable to plot.

-o, --out Output file to write the plot figure to.

4.4 Cate Desktop (GUI)

Applies to Cate Desktop, version 2.0.0-dev15

4.4.1 Overview

Cate Desktop is a desktop application and is intended to serve as the primary graphical user interface (GUI) for the CCI Toolbox.

It provides all the Cate CLI and almost all Cate Python API functionality through a interactive and user friendly interface and adds some unique imaging and visual data analysis features.

The basic idea of Cate Desktop is to allow access all remote CCI data sources and calling all Cate operations through a consistent interface. The results of opening a data source or applying an operations is usually an in-memory dataset representation - this is what Cate calls a *resource*. Usually, a resource refers to a (NetCDF/CF) dataset comprising one or more geo-physical variables, but a resource can virtually be of any (Python) data type.

The Cate Desktop user interface basically comprises *panels*, *views*, and a *menu bar*:

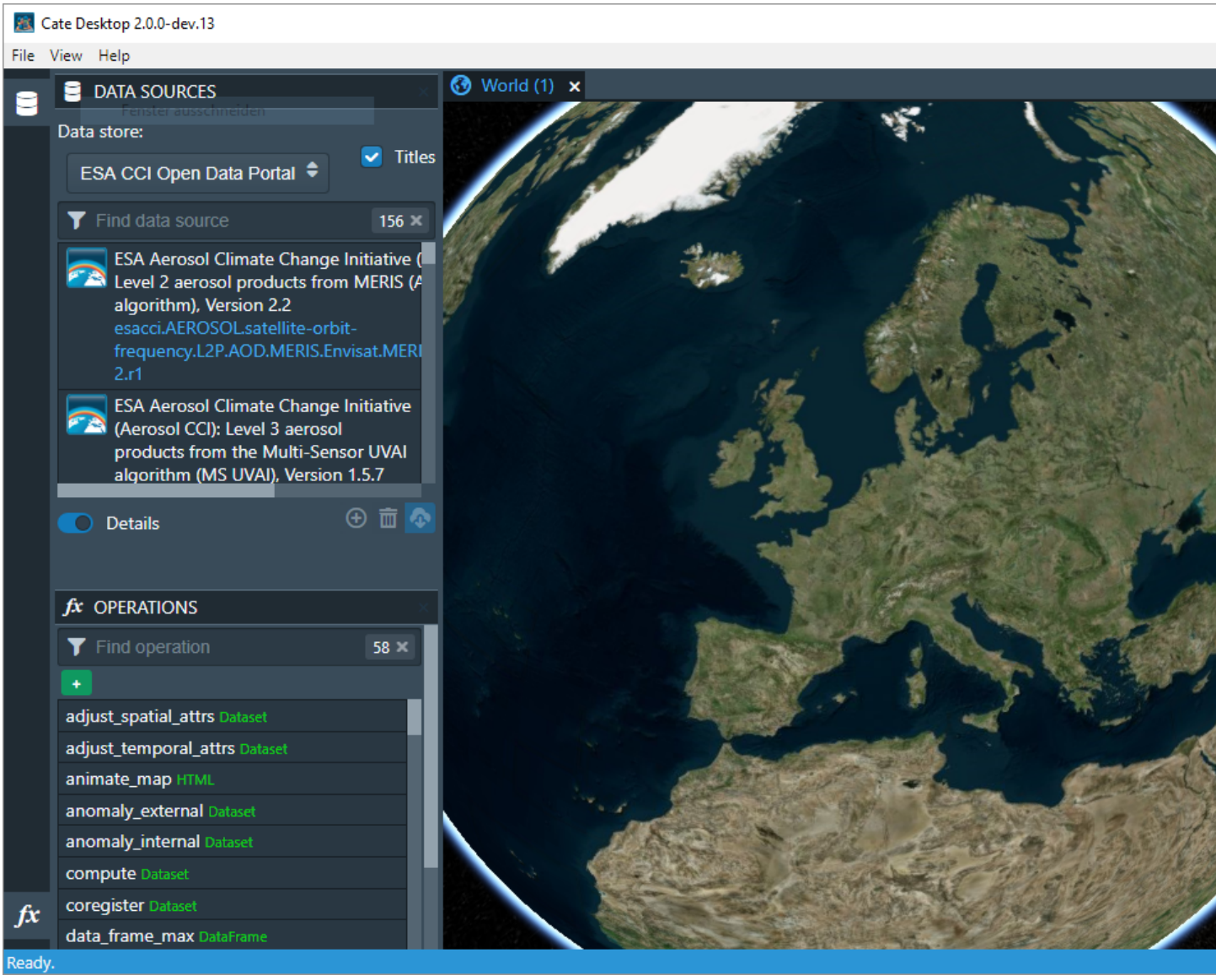


Fig. 4.6: Cate Desktop initial layout

Panels

When run for the first time, the initial layout and position of the *panels*, as shown in Fig. 4.6, reflects what just has been described above with respect to data sources, operations, resources/datasets, and variables:

1. On the upper left, the **DATA SOURCES** panel to browse, download and open both local and remote data sources, including data from ESA CCI Open Data Portal;
2. On the lower left, the **OPERATIONS** panel to browse and apply available operations;
3. On the upper right, the **WORKSPACE** panel to browser and select available resources and workflow steps resulting from opening data sources and applying operations;
4. On the lower right, the **VARIABLES** panel to browse and select the geo-physical variables contained in the selected resource.

Other panels are initially hidden. They are

- On the upper right, the **LAYERS** panel, to manage the imagery layers displayed on the active *World view*;
- On the upper right, the **PLACEMARKS** panel, to manage user-defined placemarks, which may be used as input to various operations, e.g. to create time series plots;
- On the lower right, the **STYLES** panel, to adjust the styles of the selected layer or entity;
- On the lower right, the **VIEWS** panel, to display and edit properties of the currently active view. It also allows for creating new *World views*;
- On the lower right, the **TASKS** panel, to list and possibly cancel running background tasks.

Each panel's visibility can be controlled by left- and right-most panel bars. Click on a panel icon to toggle its visibility. Between two panels, there are invisible, horizontal split bars. Move the mouse pointer over the split bar to see it turning into a split cursor, then drag to change the vertical split position. In a similar way, there are invisible, vertical split bars between the tool panels and the views area. Move the mouse cursor over them to find them.

Views

The central area is occupied by *views* that can be arranged in rows and columns. Cate currently offers three view types:

- The **world view**, displaying imagery data originaling from data variables and placemarks on either a 3D globe or a 2D map;
- The **table view**, displaying tabular resource and variable data in a table;
- The **figure view**, displaying plots from special figure resources resulting from the various plotting operations.

There may be multiple views stacked in a row of tabs, where each tab represents a view. One view within a tab row is selected and visible. The selected view can be split horizontally or vertically by dedicated icon buttons on the right of the tab row header. A split view can be stacked again by the drop down menu (...) on the right-most position of the the row tab header.

There is always a single *active view* indicated by the blueish view header text. To activate a view, click its header text. The active view provides a context for various commands, for example all interactions with the **LAYERS** and **VIEW** panels are associated with the active view.

Initially, a single World view is opened and active.

Menu Bar

Cate's menu currently comprises the **File**, **View**, and **Help** menus. The **File** menu comprises *Workspace*-related commands and allows setting user **Preferences**:

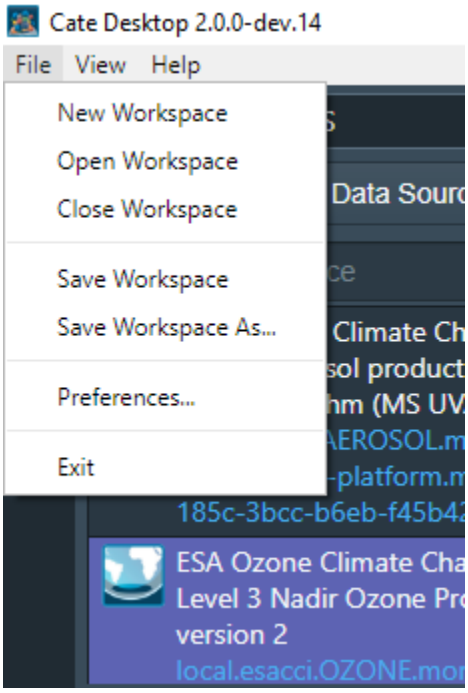


Fig. 4.7: Cate Desktop's File menu (Windows 10)

Menu item	Description
New Workspace	Creates a new *scratch* workspace. Scratch workspaces are not-yet-saved workspaces.
Open Workspace	Opens an existing workspace. Will open a dialog to select a workspace directory.
Close Workspace	Close current workspace and create a new scratch workspace.
Save Workspace	Save current workspace it its directory. Will delegate to **Save Workspace As** if it hasn't been saved before.
Save Workspace As	Opens a dialog to choose a new *empty*, directory in which the current workspace data will be saved. This will become the current workspace directory.
Preferences	Opens a dialog where users can adjust various settiungs according to their preferences. See also *Preferences Dialog*.
Exit / Quit	Exits the application

More information regarding workspaces can be found in section *Workflows, Resources, and Workspaces*.

4.4.2 Reference

Index

- *DATA SOURCES Panel*
- *OPERATIONS Panel*
- *WORKSPACE Panel*

- *VARIABLES Panel*
- *PLACEMARKS Panel*
- *LAYERS Panel*
- *STYLES Panel*
- *VIEW Panel*
- *TASKS Panel*
- *Preferences Dialog*

DATA SOURCES Panel

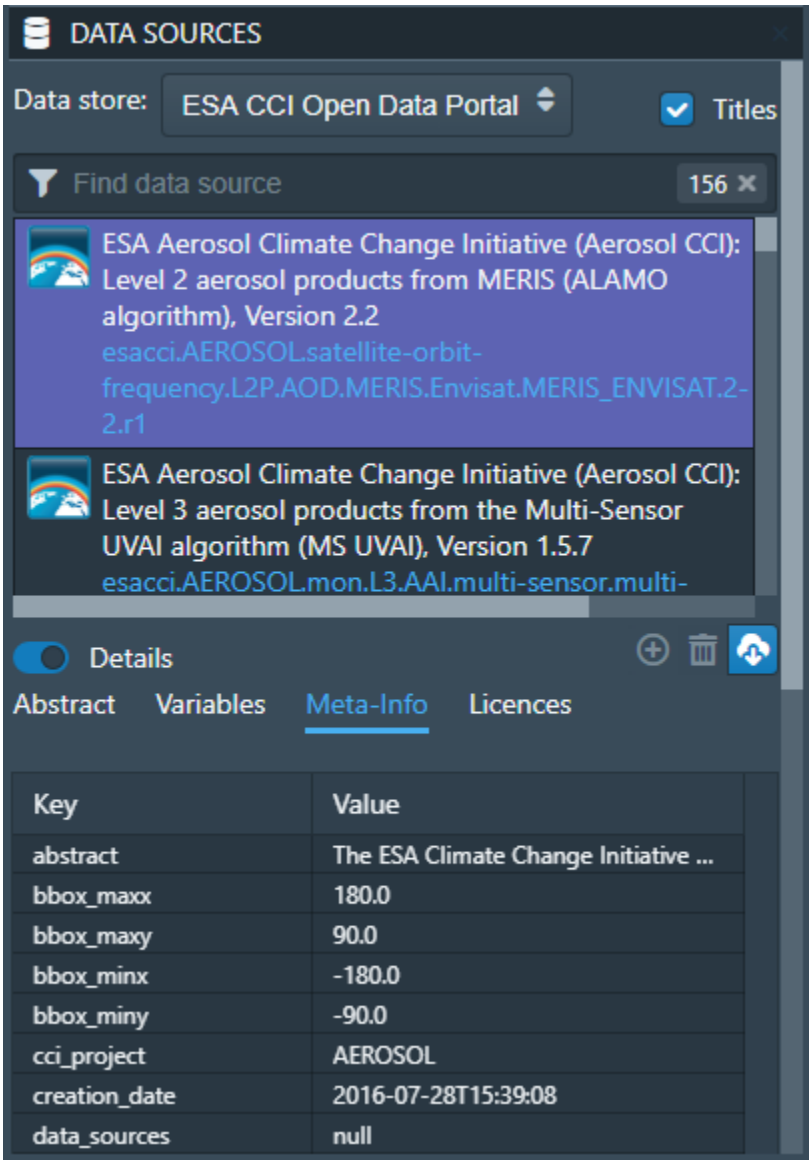


Fig. 4.8: Data Sources panel for `ESA CCI Open Data Portal`

The **DATA SOURCES** panel is used to browse, download and open both local and remote data sources published by *data stores*.

Using the drop-down list located at the top of the panel, it is possible to switch between the the currently available data stores. At the time of writing, two data stores were available in Cate, the remote *ESA Open Data Portal*, and *Local Data Sources* representing datasets made available through your file system. Below data stores selector, there is a search field, while typing, the list of data sources published through the selected data store is narrowed down. Selecting a data source entry will allow displaying its **Details**, namely the available (geo-physical) variables and the meta-data associated with the data source.

In order to start working with remote data from the *ESA CCI Open Data Portal* data store, there are two options which are explained in the following:

1. Download the complete remote dataset or a subset and make it a new *local* data source available from the local data store. Open the dataset from the new local data source. **This is currently the recommended way to access remote data** as local data stores ensure sufficient I/O performance and are not bound to your internet connection and remote service availability.

2. Open the remote dataset without creating a local data copy. **This option should only be used for small subsets of the data**, e.g. time series extractions within small spatial areas, as there is currently no way to observe the data rate and status of data elements already transferred. (Internally, we use the OPeNDAP service of the ESA CCI Open Data Portal.)

After selecting a remote data source, press the **Download** button to open the *Download Dataset** dialog shown in Fig. 4.9 to use the first option.

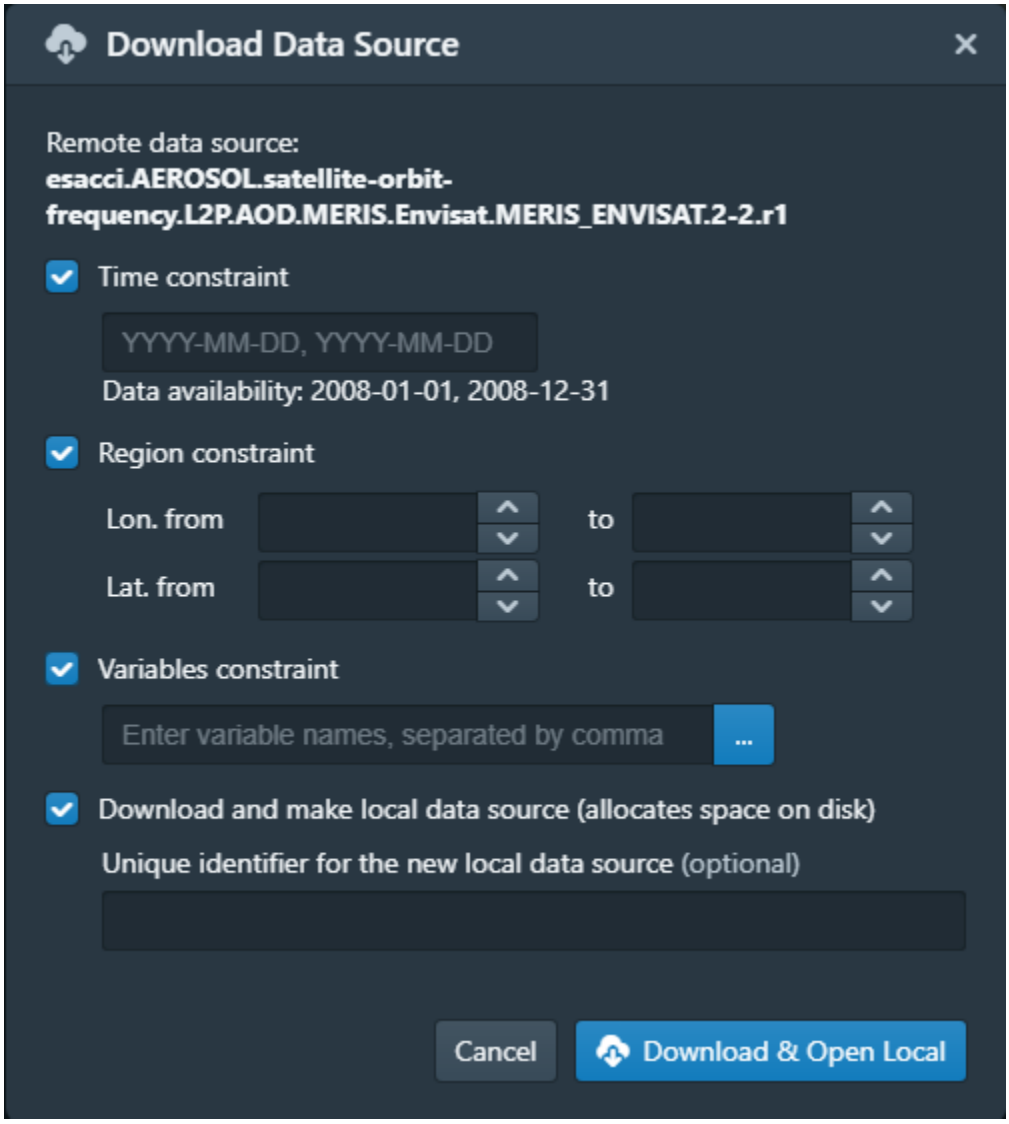


Fig. 4.9: Download Dataset dialog

Here you can specify a number of optional constraints to create a local data source that is a subset of the original remote one. You can also provide a name for the new data source. By default, the original name will be used, prefixed by `local.`.

Note: We strongly recommend to set the constraints to limit the overall amount of data to be downloaded and stored.

We are currently not able to pre-compute the amount of data and the time it will take to fully download it. Also note, that downloading remote data may require a lot of free space on your local system. By default, Cate stores this data in the user's home directory. On Linux and Mac OS, that is `~/.cate/data_stores`, on Windows it is `%USER_PROFILE%\.cate\data_stores`. Use the *Preferences Dialog* to set an alternative location.

After confirming the dialog, a download task will be started, which can be observed in the **TASKS** panel. Once the download is finished, a notification will be displayed and a new local data source will be available for the `local data store`.

To choose the second option described above, press the **Download** button to open the **Download Dataset** dialog, and then uncheck **Download and make local data source (allocates space on disk)** as shown in *Open Remote Dataset dialog*.

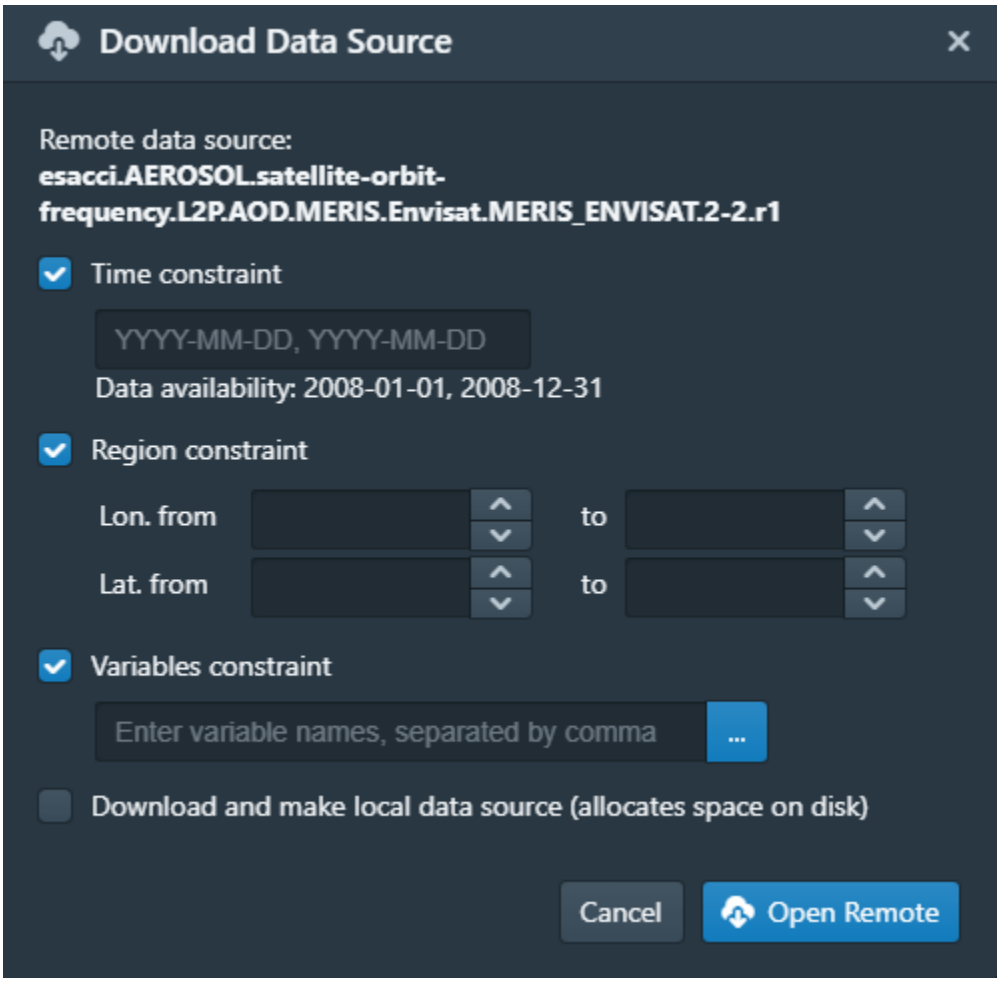
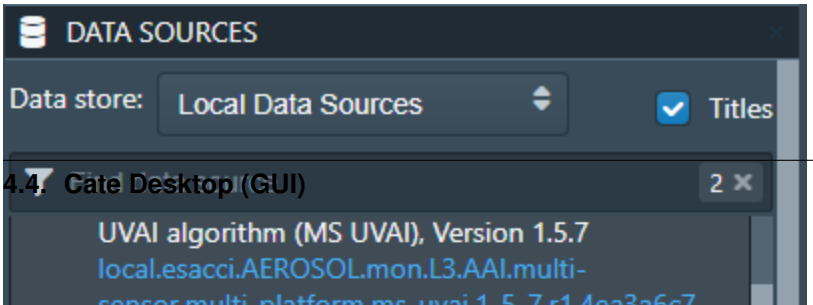


Fig. 4.10: Open Remote Dataset dialog

It provides the same constraint settings as the former download dialog. After confirming the dialog, a task will be started that directly streams the remote data into your computer's local memory. If the open task finishes, a new dataset *resource* is available from the *WORKSPACE Panel*.



Switching the data store selector to *Local Data Sources* lists all currently available local data sources as shown in

Fig. 4.11. These are the ones downloaded from remote sources, or ones that you can create from local data files.

Press the **Add** button to open the **Add Local Data Source** dialog that is used to create a new local data source. A data source may be composed of one or more data files that can be stacked together along their *time dimension* to form a single unique multi-file dataset. At the time of writing, only NetCDF (`*.nc`) data sources are supported.

Pressing the **Open** button will bring up the **Open Local Dataset** dialog as shown in Fig. 4.12 below:

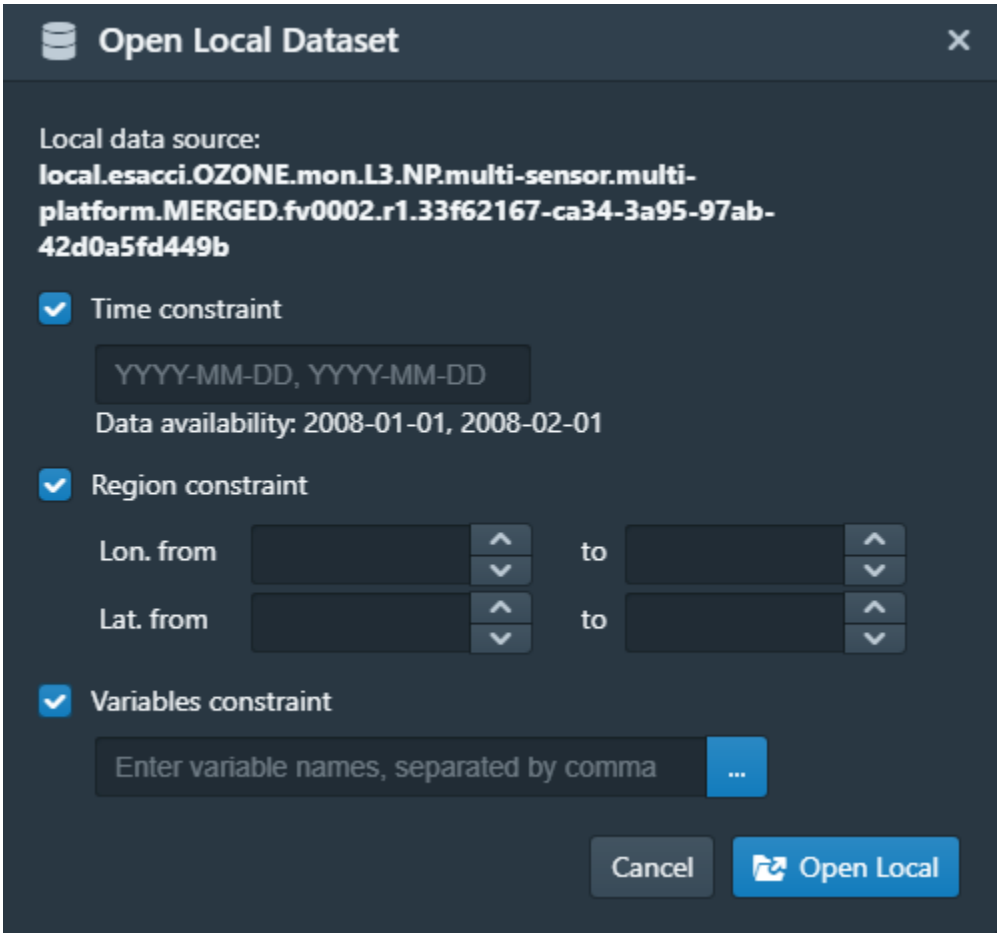


Fig. 4.12: Open Local Dataset dialog

Confirming the dialog will create a new in-memory dataset *resource* which will be available from the **WORKSPACE** panel as shown in Fig. 4.17.

Note, that **Cate will load into memory**

only those slices of a dataset, which are required to perform some action. For example, to display an image layer on the 3D Globe view, Cate only loads the 2D image for a given time index, although the dataset might be composed of multiple such 2D images that form a time series and / or a stack of atmospheric layers.

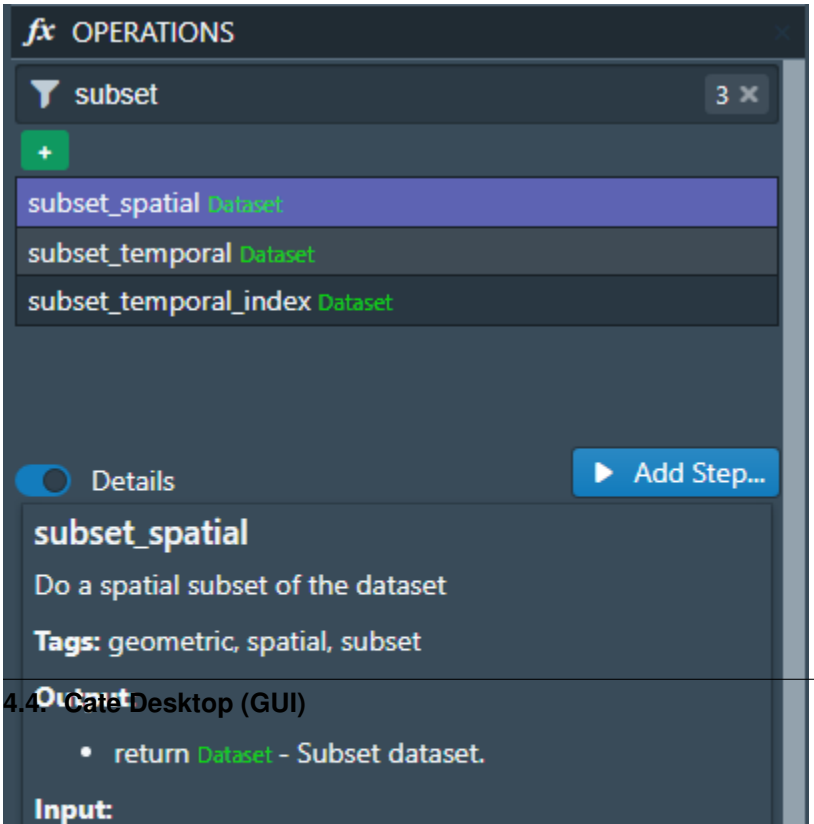
OPERATIONS Panel

The **OPERATIONS** panel is used to browse and apply available operations. The term *operations* as used in the Cate context includes functions that

- read datasets from files;
- manipulate these dataset;
- plot datasets;
- write datasets to files.

The **Details** section provides a description about the operation including its inputs and outputs.

Note: To programmers: At the time of writing, all Cate operations are plain Python functions. To let them appear in Cate's GUI and CLI, they are annotated with additional meta-information. This also allows for setting specific operation input/output properties so that specific user interfaces for a given operation is genereted on-the-fly. You might be interested to take a look at the various functions in the modules of the cate.ops Python package of Cate. These functions all use Python 3.5 *type annotations* and Cate *decorators* `@op`, `@op_input`, `@op_output` to add that meta-information to turn it into Cate *operations*.



Pressing the **Apply...** button will bring up a dialog that let's you enter the operation's parameter values. For most parameter types (numeric, boolean, text), an input field is provided. For the ones that don't have a dedicated input field, a *resource selector* is provided that let's you select a *resource* from a drop-down list. Only resources are listed whose data type match the required parameter type. Most commonly, these will be resources of type

- `Dataset`: N-dimensional, gridded data as originating from NetCDF file sets or OPeNDAP services
- `DataFrame`: two-dimensional, tabular data from CSV files

- `GeoDataFrame`: similar to `DataFrame` but include geometry data and are originating from ESRI Shapefiles and GeoJSON services.

Note that every parameter value can be set to a resource by checking the switch to right of the parameter field. This will exchange the input field by a resource selector.

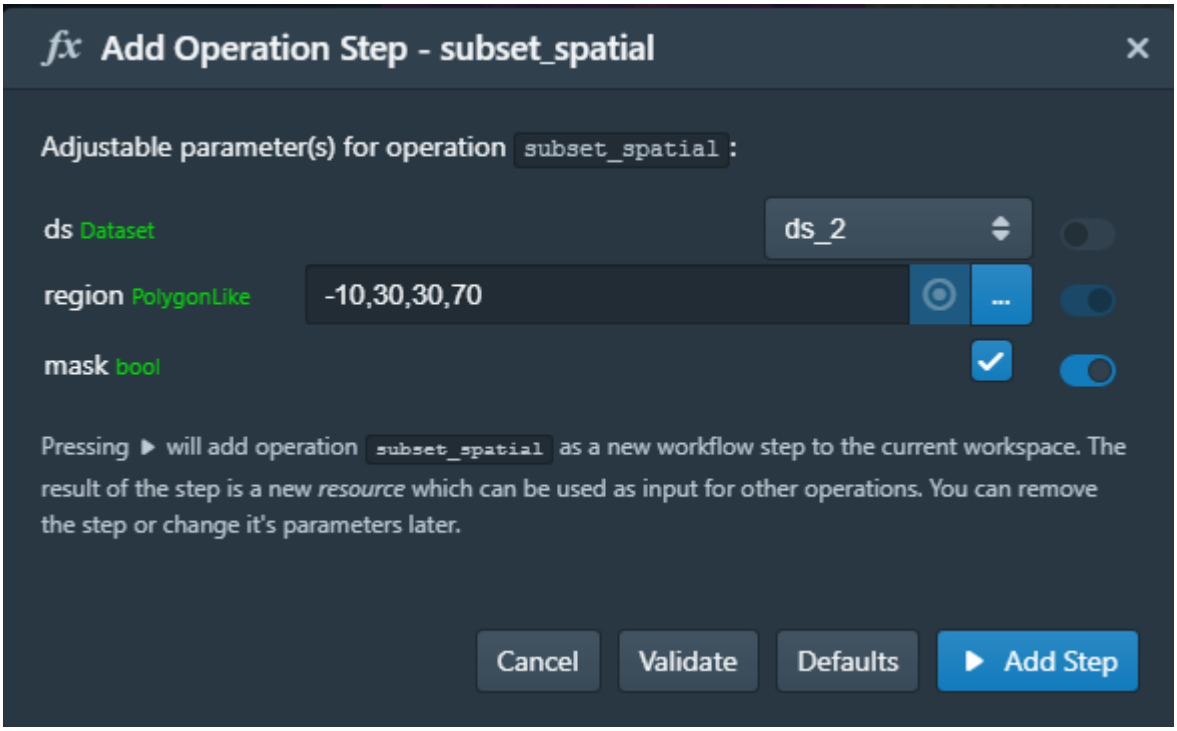


Fig. 4.14: New Operation Step dialog

After pressing the **Apply** button, the operation is being invoked and a new *workflow step* will be added to the workspace. For any operations returning a value a new *resource* will be added as well.

The new *workflow step* and the new *resource*, if any, are shown in the **WORKSPACE** panel.

Note: Some operations allow or require entering a path to a file or a directory location. When you pass a relative path, it is meant to be relative to the current workspace directory.

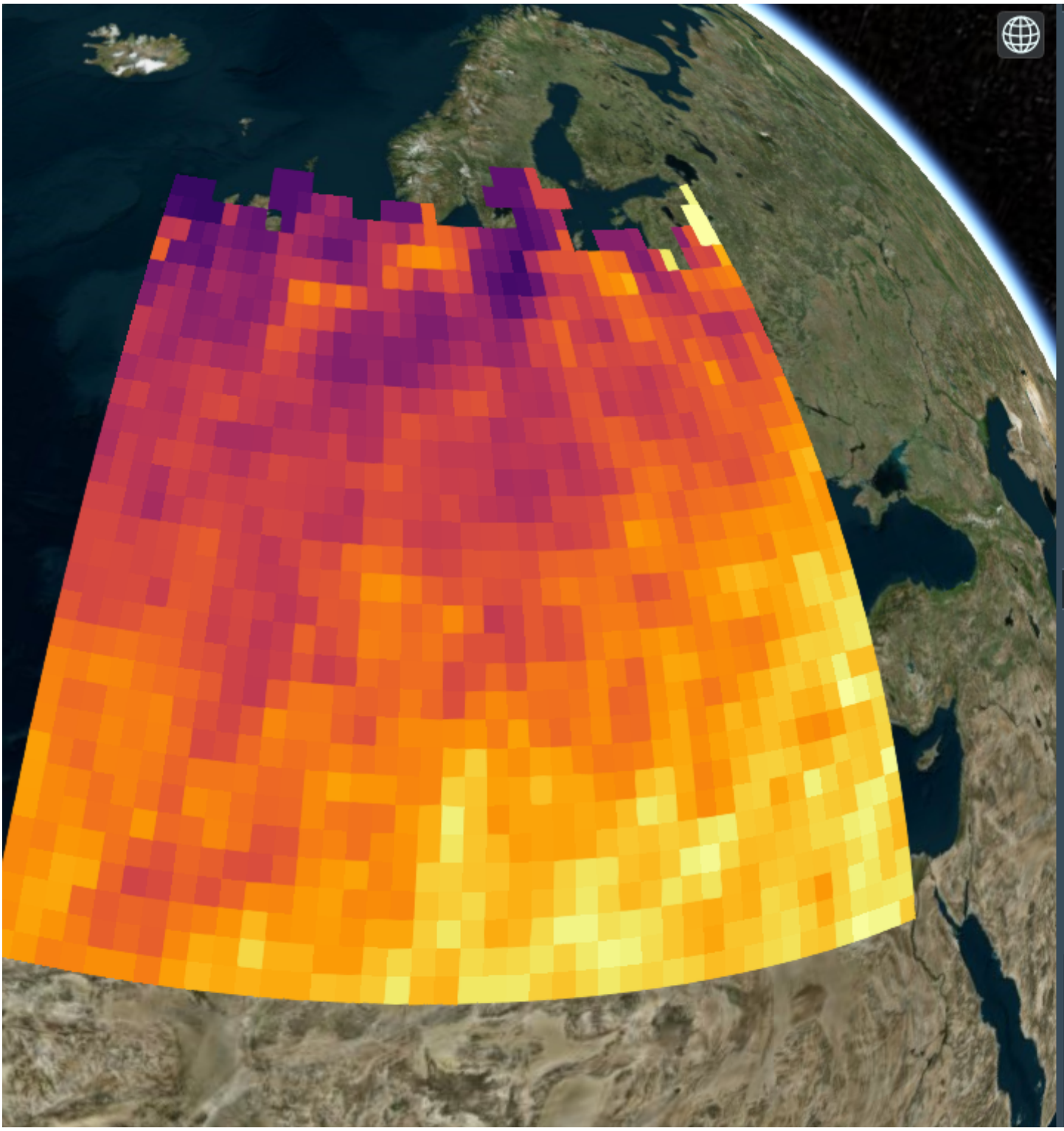


Fig. 4.15: New Operation Step in WORKSPACE Panel

WORKSPACE Panel

The **WORKSPACE** panel is used to manage the current Cate workspace whose name is displayed in the header line of the panel. To the right of the workspace name there is an indicator whether the workspace is modified or not.

In the upper left of the panel are two tools buttons that allow for * opening the workspace directory in your operating system's file explorer; * copying the workspace workflow into the operating system's clipboard as Python script, shell script or in JSON format.

The *workflow steps* and *resources* of the current workflow are shown in the respective **Workflow** and **Resources** sub-panels.

Workspace / Workflow Panel

This panel lists all the workflow steps originating from opening datasets and applying operations in chronological order. The **Details** section displays the used parameter values of a selected workflow step.

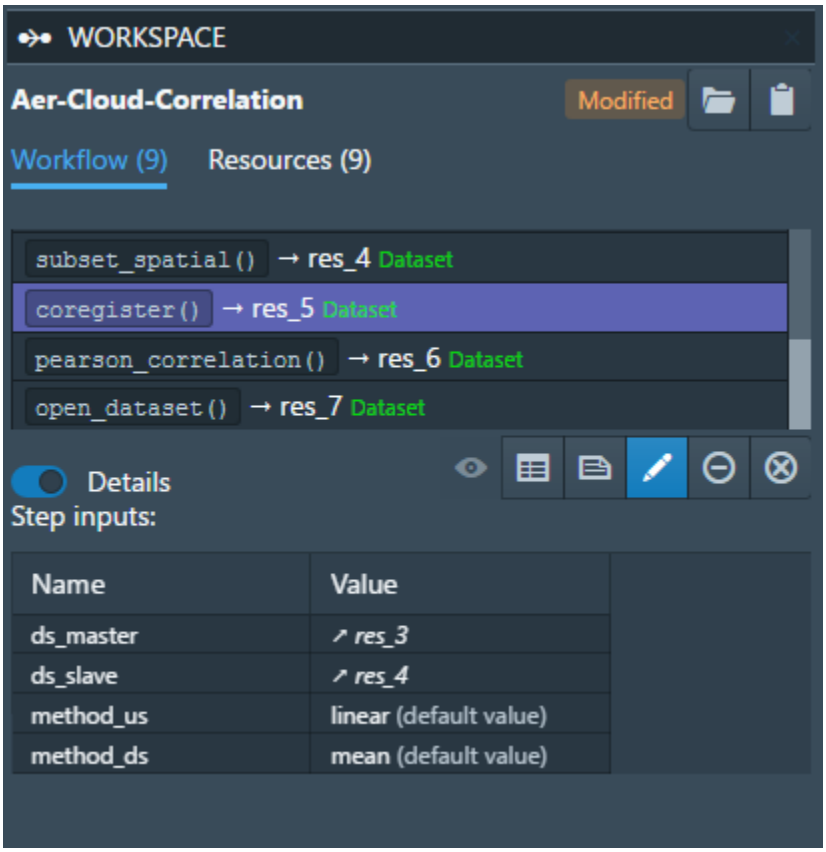


Fig. 4.16: Workspace / Workflow panel

Workspace / Resources Panel

This panel lists all the data resources originating from workflow steps. The **Details** section displays the properties and metadata of the selected data resource.

A data resource may contain any number of *data variables*. This is usually the case for any resource of type `Dataset` or `DataFrame`. The contained variables of a selected data resource are shown in the **VARIABLES** panel.

The toolbar to the lower right of the list of workflow steps or resources offers the following functions (in order):

- Show figure. Shows the associated data **resource in a figure view**. Only enabled if the resource is of type `Figure` which is the is for example the case for the various `plot_<type>()` operations.
- Show table: Shows the associated data **resource in a table view** if it is two-dimensional data.
- Edit resource / workflow step properties: Brings up a dialog

which lets you **rename a resource** and **make a resource persistent** within the workspace. The latter can drastically speed up workspace loading especially for data resources that are expensive to recompute.

- **Edit operation parameters** of a selected workflow step or resource: Brings up a the **Edit Operation Step** dialog similar to the *New Operation Step dialog*. Confirming the dialog by pressing **Apply** will invoke workflow step and compute a new resource value. All workflow step that depend on this resource will also be executed again possibly triggering other workflow step executions.
- **Remove** a selected workflow step or resource. Removal will fail if other steps depend on it.
- **Clean** the current workspace which will remove all steps and resources.

VARIABLES Panel

The **VARIABLES** panel lists the data variables of a selected resource in the **WORKSPACE** panel. The list entry shows the variable's name and its data type. When available, the value of each variable of the selected layer will be displayed next to its name after placing the mouse cursor at a point on the globe for ~600ms.

The toolbar to the lower right of the list of variables offers the following functions (in order):

- **Toggle layer visibility**: if the variable can be displayed as an image layer in the 3D globe view.
- **Add new image layer**: adds the selected variable as an image layer to the active world view, if any.
- **Create time series plot** from selected placemark. Adds a new workflow step which calls the `plot()` operation.
- **Create histogram plot**. Adds a new workflow step which calls the `plot_hist()` operation.
- **Show data in table view**. Displays 2D variables of type `DataFrame` in a table view.

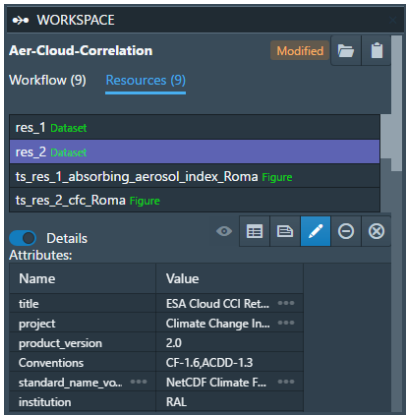
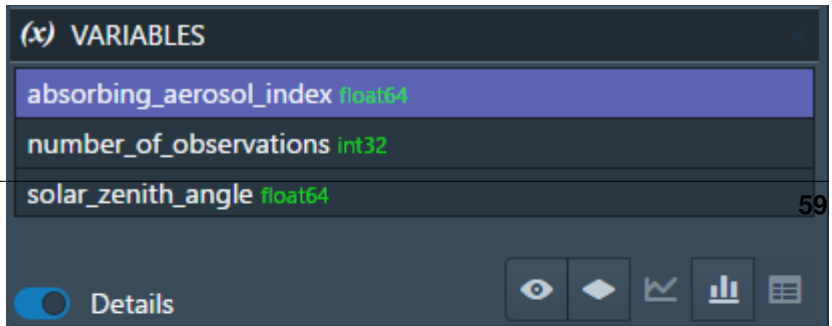


Fig. 4.17: Workspace / Resources panel

LAYERS Panel

This panel manages the list of visual layers displayed by the currently active 2D map or 3D world view. Any



number of layers can be added to active view. Two are always available:

- Selected Variable
- Country Borders

The layer *Selected Variable* displays the data of any selected variable in the **VARIABLES** panel if it is gridded and has at least the longitude and latitude dimensions (names `lon` and `lat`). The toolbar to the lower right of the layer list offers the following functions (in order):

- Add a new layer (currently you can add layers for other variables available in your workspace)
- Remove the selected layer
- Move selected layer up to render it on top of others
- Move selected layer down so other layers are rendered on top of it

The **Details** of the **LAYERS** panel lists several layer settings:

- *Data selection* with this configuration one can quickly browse through the dataset based on the layer index.
- *Layer split* with this setting, user can create a split line with one side of the line showing the globe with the selected layer and the other side showing only the globe.

PLACEMARKS Panel

This panel manages a list of placemarks - points, lines, polygons, or boxes that have a name and a geographical coordinate. Placemarks can be used to create time series plots and to extract data at a given point or area. The toolbar to the lower right of the list of placemarks offers the following functions (in order):

- Add a new marker
- Add a new polyline
- Add a new polygon
- Add a new box

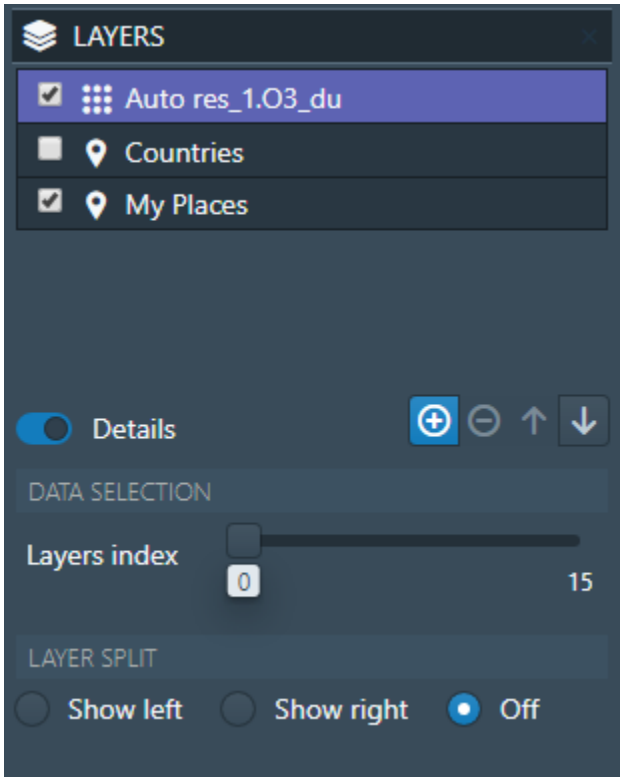


Fig. 4.19: Layers Panel

- Remove a selected placemark
- Locate the selected placemark on the map
- Copy name and/or coordinates of selected placemark to clipboard

  In addition to these buttons, there is also a Details toggle button to display or allow modification of the selected placemark. What can be modified depends on which type of placemark is selected.

  To add a new marker, click **New marker** button (the left-most), and then click any point on the Globe. A new entry is added to the list of placemarks in Placemarks Panel. When the Details toggle is enabled, you can modify the name and coordinates (in longitude and latitude) of this marker.

  To add a new polyline, click **New polyline** button (the second left-most). Click a point in the Globe to start the line, and then click the

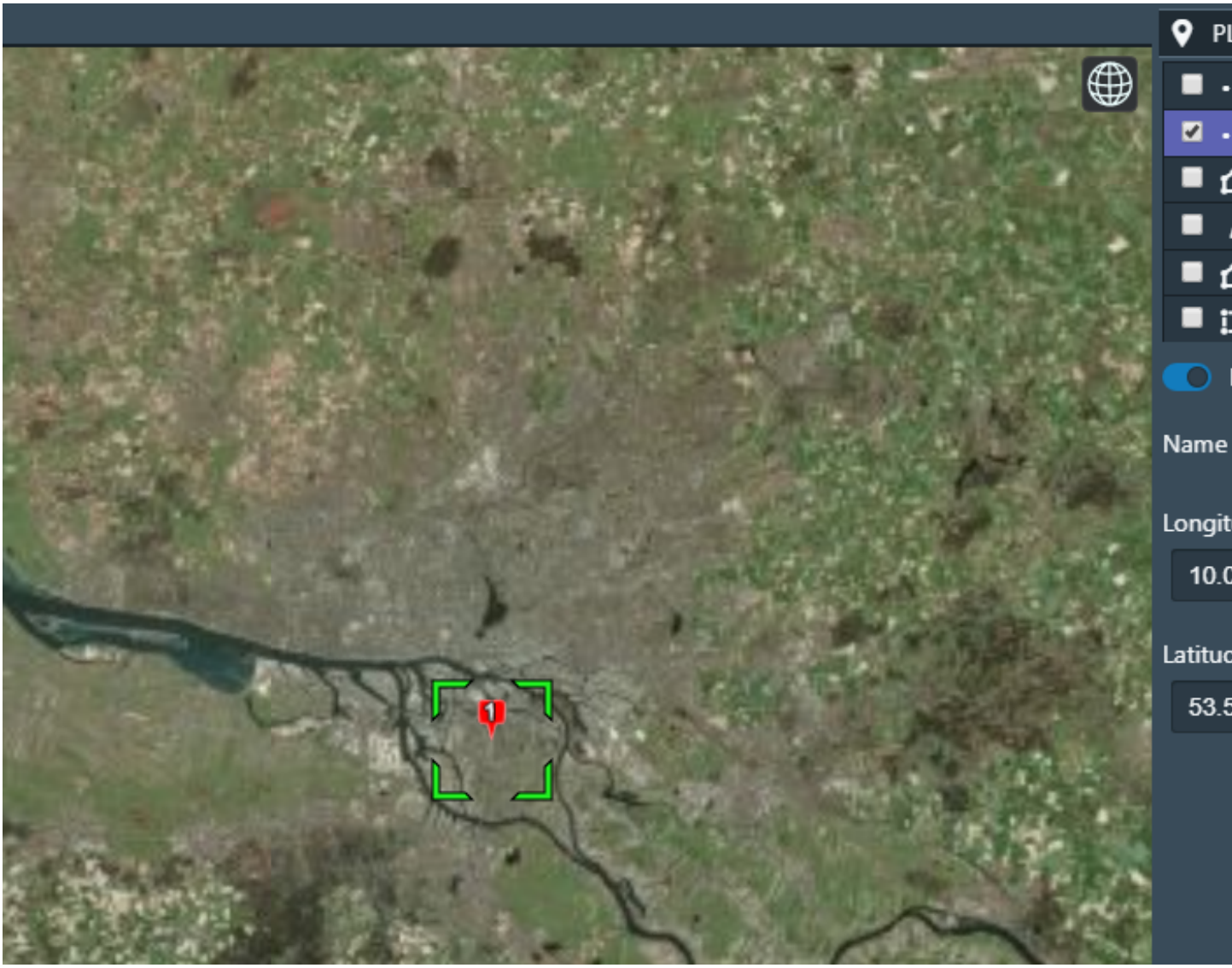


Fig. 4.20: Placemarks Panel - Marker details

next n-lines as you wish. To finish, double-click at your final point. When the Details toggle is enabled, you can modify the name of this polyline.

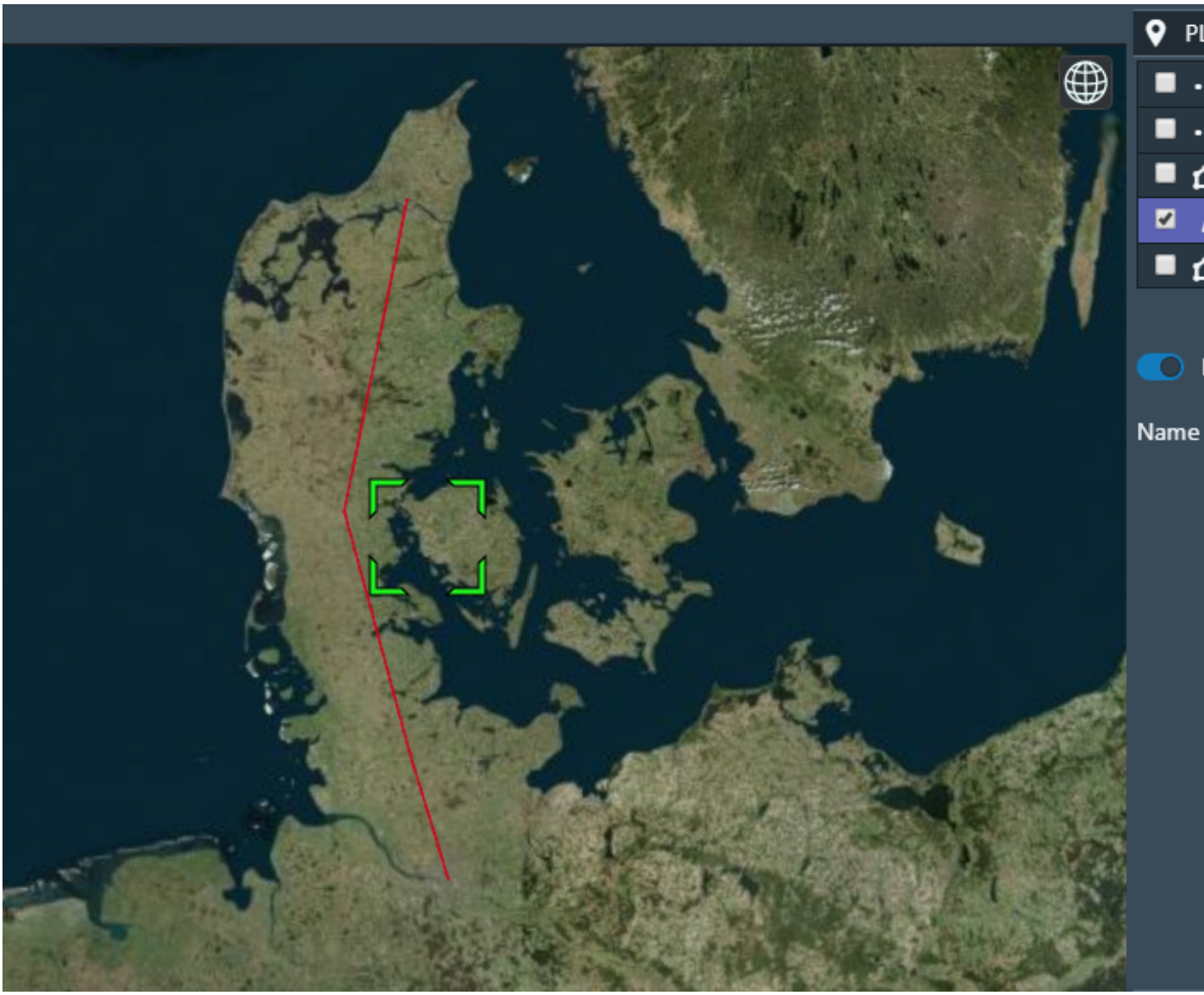


Fig. 4.21: Placemarks Panel - Polyline details

To add a new polygon, click the **New polygon** button (the third left-most). As when creating a polyline, click a point in the Globe to start the line, and then click the next n-lines as you wish. To finish, double-click at your final point. When the Details toggle is enabled, you can modify the name

of this polygon.

Fig. 4.22: Placemarks Panel - Polygon details

To add a new polygon, click the **New box** button (the fourth left-most). To start, click a point in the Globe. This will be one of the vertices of the box you are going to create. Drag it to satisfy the region you desire, and click once more to confirm the box selection. When the Details toggle is enabled, you can modify the name of this box.

The list of placemarks is currently

Fig. 4.23: Placemarks Panel - Box details

stored as a GeoJSON entry in `.cate/preferences.json` in the users home directory and restored for every Cate Desktop session.

To copy the selected placemark to clipboard, click the right-most button. There are three options how the selected placemark can be represented in three different formats: CSV, WKT, and GeoJSON.

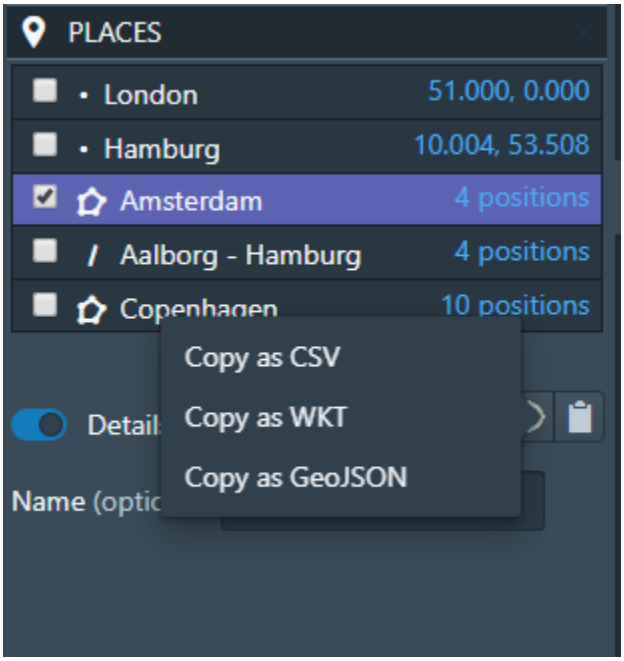


Fig. 4.24: Placemarks Panel - Copy to clipboards

STYLES Panel

This panel manages styles that can be applied to the selected layer. It has two different modesdepending on whether an image or a vector layer is selected. Here are the available settings for a vector layer:

- *Fill* controls the fill colour and the opacity of a polygon or a box.
- *Stroke* controls the width, colour, and opacity of the lines surrounding the polygon or the box.
- *Marker* controls the colour, size, and caption of the placemark. The symbol can be either a single digit of number,

a letter, or any valid **Maki** identifier (more information here)

And here are the available settings for an image layer:

- *Display Range* is the value range to which a given colour map is mapped.
- *Colour bar* is applied to gridded variables.
- *Alpha Blending* is used to mask/fade out the lower half of the display range. With *Alpha Blending* switched on, the minimum value of the display range corresponds to full transparency while opacity increases until half of the display range is reached.
- For any extra dimension of a variable that is not latitude and longitude, an *Index into <Dimension>* slider is displayed and can be used to selected the dimension's index to be displayed as layer.
- The *Opacity* controls the opacity of the selected layer
- Various *Image Enhancement* settings, like *Brightness*, * Contrast*, *Hue*.

VIEW Panel

The **VIEW** panel shows the settings of the currently active *View*. The settings depend non the type of the active view.

World Views have the following settings:

- Whether to use a 2D map or 3D globe.
- The projection for the 2D map.
- Whether to show layer titles (currently 3D globe only).
- Whether to split the current layer (currently 3D globe only).

Figures Views don't provide any special settings yet. However, in future releases, you will be able to change plot styles and size.

Table Views also don't provide any special settings yet. However, in fu-

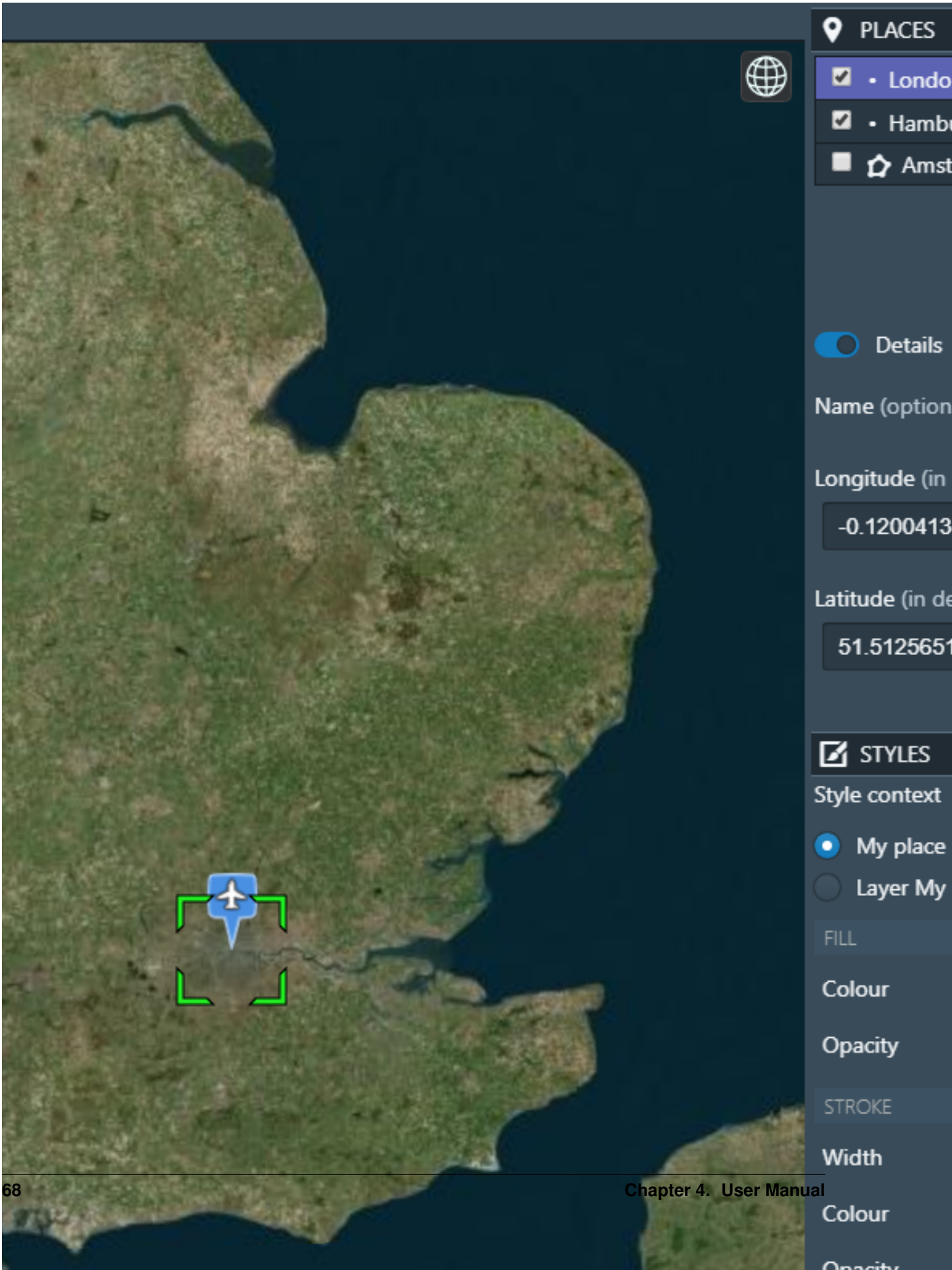

PLACES
Londo
Hambu
Amst
Details
Name (option
Longitude (in
-0.1200413
Latitude (in de
51.5125651
STYLES
Style context
My place
Layer My
FILL
Colour
Opacity
STROKE
Width
Colour

Fig. 4.26: Styles Panel for styling a polygon/box

ture releases, you will be able to specify the subset of the data ypou want to see in the table.

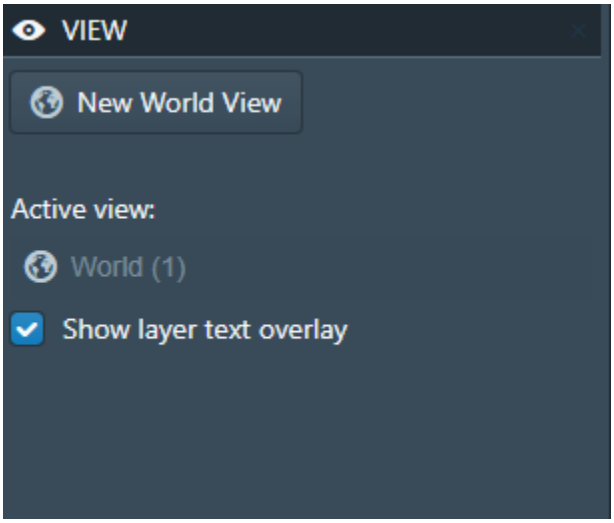


Fig. 4.27: View Panel

TASKS Panel

The **TASKS** panel shows all active tasks. Long running tasks are usually originating from downloading datasets or performing operations on datasets. Some running tasks may be cancelled, others not.

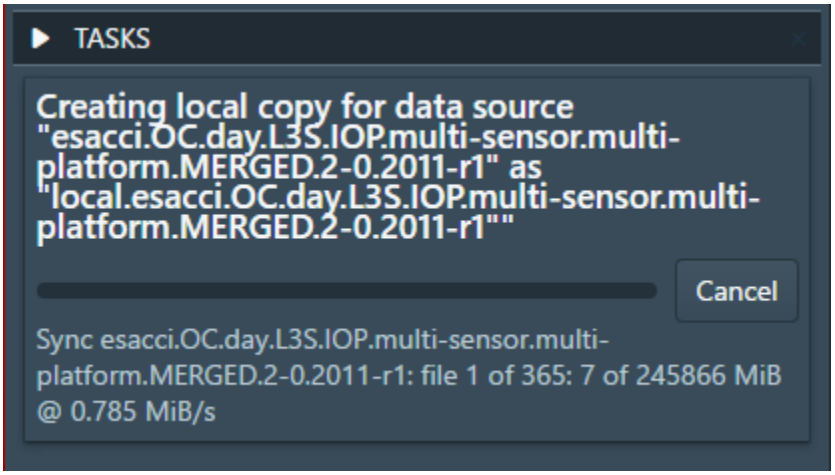


Fig. 4.28: Tasks Panel

Preferences Dialog

On the **General** tab you can specify the following settings:

- Whether to *reopen the last workspace on startup* of Cate
- Whether to automatically update the software once a newer version is available
- Whether to *open a plot view for new Figure resources*. If selected and a newly created resource is of type `Figure`, a plot view will be opened automatically. Note, `Figure` resources are created by operations named `plot_<type>()`.
- Whether to *force offline mode* after restart. In this mode Cate does not rely on an internet connection. Therefore the background satellite imagery used for the 2D/3D maps falls back to a static, low resolution map.

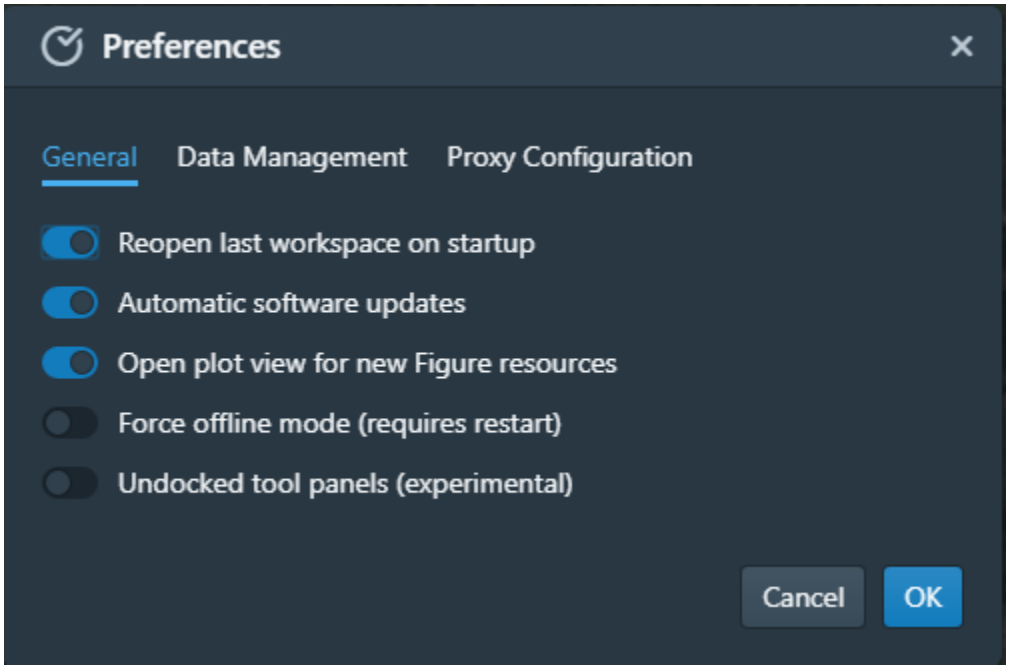


Fig. 4.29: Preferences Dialog / General

On the **Data Management** tab you can specify the following settings:

- The location of the *synchronisation directory for remote data store files*. This directory is used by Cate for

downloading and synchronizing remote data. The location shall ensure sufficient disk space for your type of application and the amount of data required locally.

- Whether to use a *per-workspace imagery cache* which may speed up image display performance. The cache is placed in each workspace directory and requires extra (disk) space.
- The *resource name prefix* which will be used by default for new resources originating from opening datasets or executing operations.

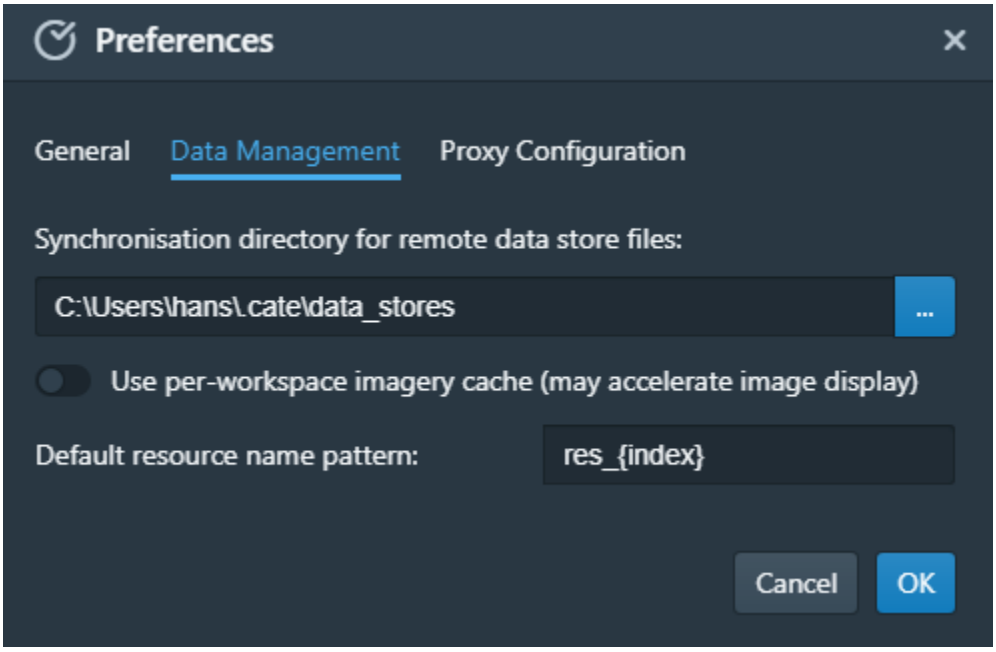


Fig. 4.30: Preferences Dialog / Data Management

On the **Proxy Configuration** tab you can specify the proxy URL if required.

4.5 Cate Python API

Python programmers can use Cate's Python API by two means

1. installing the `cate` package directly from sources
2. installing the `cate` package as a

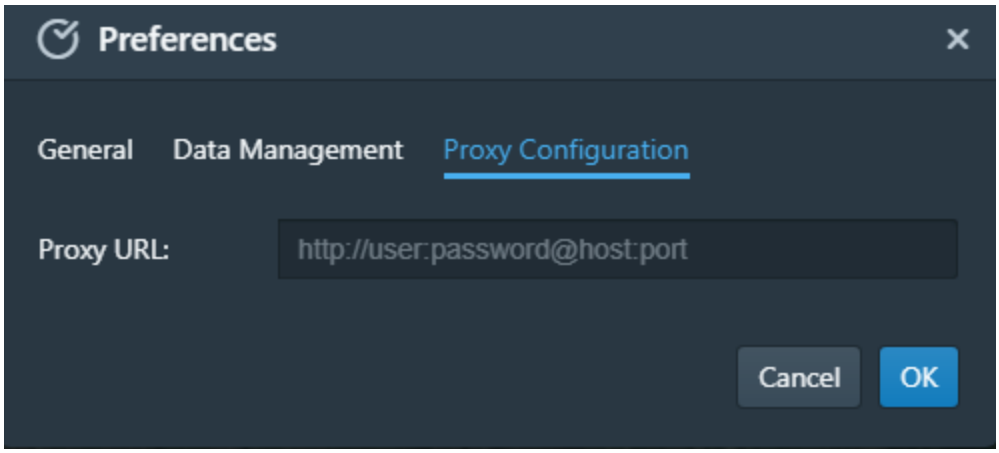


Fig. 4.31: Preferences Dialog / Proxy Configuration

Conda package into their Miniconda3/Anaconda3 environment.

Note, Cate is not yet available through *pip* (i.e. from PyPi, the common Python Package Index).

For detailed instruction, please follow the README in Cate's GitHub repository.

More information about the Cate Python API:

- *API Reference*
- API Example Notebooks
- *Developer Guide*

CHAPTER 5

Use Cases

Use cases provide application scenarios and requirements along which it will be demonstrated how the CCI Toolbox will be implemented and operated.

Use cases are defined for various user types and their climate questions come from diverse various application areas, see Table 5.1.

Table 5.1: User Types

Nr	User Type	Application Area
1	International climate research community	Contributing to Intergovernmental Panel on Climate Change (IPCC) scientific assessments, including climate model development, verification and data-assimilation, and scientists performing research on climate change monitoring, detection, attribution and mitigation. This includes (but is not limited to) the CCI Climate Modelling User Group (CMUG) and the Climate Research Groups (CRG) within each CCI ECV project.
2	Earth system science community	Working at a higher level than individual climate indicators, interested in Earth processes, interactions and feedbacks involving a fusion of theory, observations and models to which ECVs can play a role. This community includes, but is not exclusive to, those interested in WCRP Grand Science Challenges, climate system integrative approaches, major science themes, global change and socio-economic impact of climate change. Example potential users include the International Geosphere-Biosphere Programme (IGBP), dynamic global vegetation modellers, the Coupled Model Intercomparison Project (CMIP), and the Coupled Carbon Cycle Climate Intercomparison Project (C4MIP).
3	Climate service developers and providers	For use in the development and provision of climate services. The provision of climate services is outside the scope of the CCI programme, nevertheless the Agency aims to proactively support parties involved in the development and provision of such services
4	Earth system reanalysis community	For use in reanalysis model development, verification and data-assimilation
5	International bodies	Responsible for climate change policy making and coordination of climate change measurement, mitigation and adaptation efforts, including UNFCCC, CEOS, IPCC, and COP participants.
6	Undergraduate and postgraduate students	Academic interest in climate change. Sustained and dedicated actions to generate and disseminate a substantial volume of effective communication and educational materials on the specific subject of Earth Observation and Climate Change to a wider audience are required by the Agency. The CCI Toolbox shall support this endeavour.
7	Knowledgeable public	Access and interaction to the latest scientific data on climate change.

Each use case is introduced by a problem definition, which addresses a typical climate problem. This is followed by the required CCI Toolbox features and a sequence of single steps, how a user is expecting to use these features in the CCI Toolbox.

5.1 IPCC Support

User Types

- International climate research community
- International bodies

Problem Definition In its Summary for Policy Makers, the fifth IPCC Assessment Report [RD-2] shows four ECVs of the marine environment as indicators of a changing climate. This figure depicting the "(a) extent of Northern Hemisphere March-April (spring) average snow cover; (b) extent of Arctic July-August-September (summer) average sea ice; (c) change in global mean upper ocean (0–700 m) heat content aligned to 20062010, and relative to the mean of all datasets for1970; (d) global mean sea level relative to the 1900–1905 mean of the longest running dataset, and with all datasets aligned to have the same value in 1993, the first year of satellite altimetry data" in the form of annual values with available uncertainties expressed as shadings, could also constitute a CCI Toolbox product. For a second figure, change in sea ice extent and ocean heat content are calculated on a regional basis and contrasted with land surface temperature anomalies. Additionally, global averages of land surface, land and ocean surface temperature as well as ocean heat content changes are presented. All observational time series are compared with model output. This could have been a CCI Toolbox operation, too.

Required Toolbox Features Step 1

- Access to and ingestion of multi ESA CCI ECVs (Sea Ice, SST and Sea Level)
- Access to and ingestion of other ECV sources (ESA GlobSnow)
- Tools to perform QC on input data (at least visual checking, consistency with historic time series)
- Resampling and aggregation to a common spatio-temporal grid including propagation of uncertainties
- Comparison of sea ice coverage from Sea Ice, OC and SST (this may require own processors)
- User programmed model to derive upper ocean heat content from SST
- Aggregation to global averages including uncertainty propagation
- Line plots as output, showing means and uncertainties

Additionally Required Toolbox Features Step 2

- Access to and ingestion of further ESA data (LST from GlobTemperature) and model output (sea ice, upper ocean heat content, LST, NST)
- Band math or user programmed tool to combine SST and land surface temperature
- Spatial filtering to perform the analysis on a regional scale (e.g. using shape files)
- Ensemble statistics to show model ensemble mean and uncertainties in comparison to results based on (satellite) observations

5.2 School Seminar Climate and Weather

User Types

- Knowledgeable public

Problem Definition As a school project, measurements of air temperature, precipitation and wind speed from the school-run weather station shall be compared to long-term climate data in the form of ESA's CCI Cloud and Soil moisture climatological means. Finally, it shall be assessed if the measurements are within the climate means for the particular location.

Required Toolbox Features

- Access to and ingestion of ESA CCI Cloud and Soil Moisture data
- Access to and ingestion of user supplied data (NST, PRE, wind speed); if required programming of an interface to a measurement device
- Extraction of cloud and soil moisture time series data corresponding to the location of the school
- Calculating the climatological means from the time series including propagation of uncertainties
- Filtering of the measurement data from the meteorological station: e.g. detection of outlier or gap filling (implemented in the toolbox or programmed by the students)
- Generation of a line plot showing the CCI and the meteorological station data.
- Optional: comparison of the climatology at the school location with those from other locations on earth: selection of other locations and comparing the climatologies in one graph (i.e. without meteorological station data from the other location)

Notes This could also be a user visiting the website of a meteorological station and the website has included a widget that accesses the toolbox to perform the steps described.

5.3 Glaciers and Sea Level Rise

User Types

- International climate research community
- Earth system science community
- Earth system reanalysis community

Problem Definition A scientist wants to know: "What is the contribution of all glaciers to global sea level rise over a given time period in the future?".

Required Toolbox Features

- Access to and ingestion of ESA CCI Glacier and Sea Level data
- Access to and ingestion of all relevant in-situ measurements from the past (via WGMS)
- Access to and ingestion of topographic information for each glacier from a DEM
- Spatial and temporal aggregation, re-gridding and possibly gap filling in order to make the data fields compatible with the model grid for model calibration and validation
- Hypsometry calculation with a user-supplied plug-in (i.e. extending the toolbox, CLI, API, GIS tools)
- Spatial resampling and converting back and forth between different coordinate systems, projections and ellipsoids to match all data spatially (co-registration)
- Running of a prediction model (user-supplied plug-in or use of CLI, API), output creation (maps, graphs, tables) and comparison with validation data

5.4 Extreme Weather Climate Service

User Types

- Climate service developers and providers

Problem Definition In March 2012, the article "US heatwave may have been made more likely by global warming" by Andrew Freedman, senior science writer for Climate Central, was published in *The Guardian*. He wrote about extreme events, using the example of the increased occurrence of heat waves in March in relation to Greenhouse Gases. The article included a map of temperature anomalies over North America compared to the 2000–2001 reference period to illustrate this. Furthermore, several statements which require analysis of large data sets and time series were made. The CCI Data and CCI Toolbox could have supported this analysis.

Required Toolbox Features

- Access to and ingestion of ESA CCI GHG data
- Access to and ingestion of ESA GlobTemperature data
- Geometric adjustments
- Spatial subsetting
- Computation of statistical quantities (mean of all month/season of a reference time series and percentiles)
- Computation of anomalies
- Map generation and with a simple colour coding to present a clear message

5.5 School Seminar Glacier

User Types

- Undergraduate and postgraduate students

Problem Definition A student (at school) wants to know for a seminar paper: "What is the largest glacier in the world and how has this glacier changed in the past compared to other glacierized regions?".

Required Toolbox Features

- Access to and ingestion of the Randolph Glacier Inventory (RGI; database with contributions of CCI Glaciers) via GLIMS homepage
- Sorting for size
- Selection, extraction and saving to disk of the data for the largest glacier
- Access to and ingestion of glacier fluctuation data, e.g. from World Glacier Monitoring Service (WGMS)
- Filtering of fluctuation data according to a selection based on reference data (here the RGI data)
- Extraction of a summary of global glacier fluctuations from WGMS data base
- Data comparison (statistical values, deviations, graphs, maps, animations) and export

5.6 Teleconnection Explorer

User Types

- Undergraduate and postgraduate students

Problem Definition As part of a project on climatic teleconnection, a student investigates how El Niño-Southern Oscillation (ENSO) relates to monsoon rainfall. A result could be a plot showing the sliding correlation between Indian Summer Monsoon Rainfall (ISMR) and SST anomalies [RD-4]. A more sophisticated version of this task would be to calculate the Multivariate ENSO Index (MEI, [RD-5], [RD-6]). Additionally, also the comparison of the ENSO index with other CCI datasets (e.g. Cloud, Fire) would be interesting.

Required Toolbox Features

- Access to and ingestion of ESA CCI SST and Soil Moisture data
- Geometric adjustments
- Spatial (manually by drawing a polygon of the particular area) and temporal filtering and subsetting for both data sets
- Calculation of anomalies and statistical quantities
- Visual presentation of statistical results and time series
- ENSO index calculation from SST data (built-in function, user-supplied plug-in or CLI, API)
- Calculation of the absolute anomaly on the log transformed soil moisture data (this should be a standard function/processor provided by the toolbox)
- Calculation of the correlation between the two time series with a lag of 30 days
- Generation of a correlation map and export of the correlation data (format options) regarding the date range chosen
- Generation of a time series plot of the correlation by the selection of a location in South East Asia on the correlation map
- Saving of the image and the underlying data (format options)

In case of choosing the MEI instead of a solely SST-based index:

- Access to and ingestion of additional data for MEI (sea-level pressure (P), zonal (U) and meridional (V) components of the surface wind, sea surface temperature (S), surface air temperature (A), and total cloudiness fraction of the sky (C))
- Geometric adjustments
- Index calculation including EOF analysis (incorporated by built-in function, user-supplied plug-in or CLI, API)

Additional Features

- Access to and ingestion of additional ESA CCI data (fire, clouds, ocean colour, sea ice)
- Geometric adjustments
- Spatial and temporal filtering
- Calculation of statistic quantities and correlations
- Generation of maps and plots
- Export of the data

5.7 Regional Cryosphere Climate Service

User Types

- Climate service developers and providers

Problem Definition The Federal Office of Environment (FOEN) in Switzerland wants to provide an internet-based platform to disseminate latest information on the cryosphere and its changes in Switzerland. Such information could be, for example, the number of days with snow or other parameters like the glacier extent or start of the melting season. Before the technical work with the toolbox can be performed a user survey would be required to obtain detailed requirements for such a climate service.

Required Toolbox Features

- Access to and ingestion of RGI Glacier and WGMS fluctuation data
- Access to and ingestion of meteorological and snow cover data (from MeteoSchweiz and Institute for Snow and Avalanche Research (SLF))
- Geometric adjustments and spatial intersection
- Access to and ingestion of ESA CCI Glacier data
- Access to and ingestion of latest meteorological data
- Geometric adjustments
- Extraction of area and time period
- Generation of graphs (e.g. cumulative glacier length changes): descriptive statistical analysis (at least mean values, variances, anomalies) with user-controlled display and format options, annotations (need to load and display information on limitation and data sources)
- Decision on any other data that should be made available (e.g. more permanently, quick links)

Note The general decision on layout, data sets etc. will be taken by the FOEN outside the CCI Toolbox but will be input to the selection options.

5.8 World Glacier Monitoring Service

User Types

- International bodies

Problem Definition A service of the World Glacier Monitoring Service (WGMS) based on ESA CCI products, combined with other environmental parameters as well as meta data on glaciers, could be the provision of a database of glacier volume changes derived from remote sensing data (e.g. DEM differencing and altimetry sensors)

Required Toolbox Features

- Access to and ingestion of RGI Glacier and WGMS fluctuation data
- Access to and ingestion of ESA CCI Glacier data
- Access to and ingestion of altimetry data and glacier meta data
- Geometric adjustments
- Subsetting and filtering of data according to user defined criteria
- Data quality and consistency checks
- Search for information about persons responsible for meta data according to a list of criteria, procurement of meta data
- Adjustment of formats and metadata until they fit into the database (reference keys)
- Additional: Selection of locations, time-periods, Calculation of means, anomalies, variances

- Quality checks and data upload to the database

5.9 Relationships between Aerosol and Cloud ECV

User Types

- Earth system science community

Problem Definition A climate scientist wishes to analyse potential correlations between Aerosol and Cloud ECVs.

Required Toolbox Features

- Access to and ingestion of ESA CCI Aerosol and Cloud data (Aerosol Optical Depth and Cloud Fraction)
- Geometric adjustments
- Spatial (point, polygon) and temporal subsetting
- Visualisation of both times series at the same time: e.g. time series plot, time series animation
- Correlation analysis, scatter-plot of correlation statistics, saving of image and correlation statistics on disk (format options)

Exemplary Workflow op_specs/uc_workflows/uc09_workflow

5.10 Scientific Investigation of NAO Signature

User Types

- Earth system science community

Problem Definition A climate scientist wishes to investigate the signature of the North Atlantic Oscillation (NAO) in multiple ECVs using a processor built by another climate scientist and contributed to the toolbox.

Required Toolbox Features

- Access to and ingestion of ESA CCI ECV data (e.g. clouds, sea ice, sea level, SST, soil moisture)
- Access to and ingestion of external data (NAO time series)
- Geometric adjustments
- Spatial and temporal subsetting
- Use of externally developed plug-in to apply R [RD-7]: removal of seasonal cycles, lag-correlation analysis between each ECV and the NAO index
- Generation of time-series plot for each ECV
- Export statistics output to local disk

5.11 School Project on Arctic Climate Change

User Types

- Undergraduate and postgraduate students

Problem Definition As part of a project on Arctic climate change, an undergraduate student wishes to look at different ECVs on a polar stereographic projection.

Required Toolbox Features

- Access to and ingestion of CCI ECV data (e.g. sea ice, ice sheets, sea level, SST, clouds aerosol)
- Access to and ingestion of ECV data from external server
- Remapping to fit data onto user-chosen projection
- Spatial and temporal subsetting
- Gap-filling (user-chosen strategy)
- Generation of scalable maps

5.12 Marine Environmental Monitoring

User Types

- Climate service developers and providers
- Knowledgeable public

Problem Definition The eReef project examines the living conditions of the Great Barrier Reef via two subprojects. On the one hand, the aim of the Marine Water Quality Dashboard is to estimate water quality indicators from ocean colour data to deduce brightness and therefore the vitality of coral-competing seagrass and algae. ReefTemp Next Generation, on the other hand, seeks to assess the risk of bleaching due to overly warm water by calculating heat stress indices. This could also be a task for the CCI Toolbox environment.

Required Toolbox Features

- Access to and ingestion of ESA CCI SST and Ocean Colour data
- Access to and ingestion of data regarding brightness-plant growth relationships, competitor relationships (plant growth-coral vitality), and heat stress-coral vitality relationships.
- Geometric adjustments
- Temporal and spatial subsetting
- Implementation of a water optical property model (plug-in, CLI, API)
- Calculation of anomalies, extremes (+ trend analysis, correlations)
- Index calculation (plug-in, CLI, API)
- Visualisation, graphs, data export

5.13 Drought Occurrence Monitoring in Eastern Africa

User Types

- Climate service developers and providers
- International bodies
- Knowledgeable public

Problem Definition Due to time-lagged teleconnections, weather conditions in Eastern Africa are highly influenced by climate modes of variability in remote regions. Therefore, climate indices such as for ENSO, MJO or QBO as well as the NDVI can be used to estimate the drought probability in Africa. Long time series from satellite observations act as a basis for the construction of statistical forecasting models, which are then run by latest meteorological data.

Required Toolbox Features

- Access to and ingestion of ESA CCI SST, Clouds, Soil Moisture, and Fire data
- Access to and ingestion of non-CCI observational (e.g. NST, PRE, OLR, SLP, NDVI) and latest meteorological data
- Geometric adjustments
- Spatial and temporal subsetting (for each variable)
- NDVI and climate index calculation (ENSO, MJO, QBO indices), includes descriptive statistics
- Estimation of predictor (SST, SST gradients, OLR, cloud properties, climate indices) – predicant (NST and PRE E Africa) relationship by time-lagged (linear) regression model (plug-in, CLI, API)
- Run model by means of latest meteorological data
- Visualisation and export of results (graphs, maps, animations, tables)

5.14 Drought Impact Monitoring and Assessment in China

User Types

- Climate service developers and providers
- International bodies

Problem Definition (Solely basic idea taken from WMO (2015)) Drought occurrence and severity in terms of fire, vegetation state and soil moisture shall be estimated by the use of temperature and rainfall (+ humidity and evapo-transpiration) data to prepare countermeasures. This is achieved by the construction of an empirical statistical model using satellite-derived time series which is afterwards run by actual meteorological data.

Required Toolbox Features

- Access to and ingestion of ESA CCI Soil Moisture and Fire data
- Access to and ingestion of non-CCI NST, PRE, and NDVI observation and latest meteorological data
- Geometric adjustments
- Spatial and temporal subsetting (for each variable)
- (Descriptive statistic analysis)
- Estimation of predictor (NST, PRE) – predicant (soil moisture, vegetation state, fire occurrence) and PRE E Africa) relationship by time-lagged (linear) regression model (plug-in, CLI, API)
- Run model by means of latest meteorological data

- Visualisation and export of results (graphs, maps, animations, tables)

5.15 Renewable Energy Resource Assessment with regard to Topography

User Types

- Climate service developers and providers
- International bodies

Problem Definition The long-term potential for renewable energy generation is to be estimated by considering the effect of cloud features, aerosols, ozone and water vapour on solar irradiance as well as topographical data.

Required Toolbox Features

- Access to and ingestion of ESA CCI Ozone, Clouds, and Aerosols data
- Access to and ingestion of non-CCI data (water vapour, irradiance)
- External topographical data: preprocessed data regarding roof area, tilt, orientation from DEM
- Geometric adjustments
- Spatial and temporal subsetting
- Implementation of fast radiative transfer calculations (plug-in, CLI, API) to deduce solar irradiance
- Extraction of areas with high potential regarding solar irradiance (set appropriate boundary values)
- Extraction of areas with suitable tilt and orientation
- Visualisation of suitable areas in a map
- Estimation of Solar Power potential from pixel count
- Export of Results

5.16 Monitoring Tropical Deforestation

User Types

- Climate service developers and providers
- International bodies

Problem Definition Maps of forest cover, change and deforestation shall be produced depicting forest status and trends for 5-year periods centred around 2000, 2005, and 2010. Additionally, vector data regarding infrastructure (e.g. road works) could be obtained from local authorities and compared with forest evolution.

Required Toolbox Features

- Access to and ingestion of ESA CCI Land Cover data
- Additional: access to and ingestion of vector data regarding infrastructure
- Spatial and temporal adjustments and subsetting

- Extraction of forest class
- Estimation of forest area for multiple time-steps
- Additional: layer operations comprising infrastructure and forest data (vector and raster)
- Visualisation of forest area changes (animated?), relation to infrastructure
- Data export

5.17 Stratospheric Ozone Monitoring and Assessment

User Types

- Climate service developers and providers
- International bodies

Problem Definition As UV exposure is a highly relevant health factor, the state of the ozone layer shall be monitored as well as its influence parameters.

Required Toolbox Features

- Access to and ingestion of ESA CCI Ozone data
- Access to and ingestion of surface-based measurements of ozone-depleting substances, data regarding UV exposure
- Geometric adjustments
- Spatial (horizontal and vertical) and temporal subsetting
- Assessment of total ozone values as well as vertical profiles
- Estimation of ozone-UV exposure relationship data
- Correlation analysis between ozone values and concentrations of ozone-depleting substances
- Trend analysis of stratospheric ozone concentrations
- Visualisation (maps, graphs) and export of the results

5.18 Examination of ENSO and its Impacts based on ESA CCI Data

User Types

- Undergraduate and postgraduate students

Problem Definition As a project work, a student's task is to conduct an examination of ENSO solely by the use of ESA CCI data. For this, the first principal component of the combined EOF analysis of cloud cover, sea level and sea surface temperature in the (central/eastern) equatorial Pacific shall be intercompared with ocean colour (eastern equatorial Pacific), fire disturbance and soil moisture (landmasses adjacent to the eastern and western tropical Pacific).

Required Toolbox Features

- Access to and ingestion of ESA CCI Cloud, Fire, Ocean Colour, Soil Moisture, Sea Level, and SST data
- Temporal/spatial selections or aggregations in case of differing temporal or spatial data set resolutions

- Temporal and spatial filtering regarding time period and particular areas of interest, spatial mean values for ocean colour, fire, soil moisture (particular regional boundaries need to be assessed)
- Test for normal distribution (using plug-in/API)
- **EOF analysis:**
  - Removal of seasonal cycle and linear/quadratic trends to clarify ENSO signal
  - Conduction of EOF analysis involving array processing and statistics by means of a plug-in/API
  - Visual examination of EOF map and eigenvalues, to clarify if ENSO typical patterns are present and explained variance is sufficiently high
- Correlation statistics (different lags) between time series of first principal component and ocean colour, fire disturbance E, fire disturbance W, soil moisture E, soil moisture W including t test for the assessment of significance
- Plotting of all computed time series in one coordinate system
- Option to manually select point location on globe to compare data with PC1
- Storage of plots, time series data, correlation statistics on local disk

5.19 GHG Emissions over Europe

User Types

- Knowledgeable public

Problem Definition A person wants to know how greenhouse gas emissions over Europe evolved during the last years.

Required Toolbox Features

- Access to and ingestion of ESA CCI GHG data
- Selection of required products/variables
- Temporal and spatial subsetting
- Generation of maps/animations depicting the evolution of GHG emissions

5.20 Examination of North Eastern Atlantic SST Projections

User Types

- Climate research community

Problem Definition A climate scientist uses CCI data to validate the output of several CMIP5 models concerning SST in the north eastern Atlantic Ocean. Afterwards he picks the best model runs to perform a trend analysis regarding the future evolution using the ensemble mean and uncertainties as well as probability density functions. Applying an Analysis of Variance, he examines the different results of the models.

Required Toolbox Features

- Access to and ingestion of ESA CCI SST data
- Access to and ingestion of CMIP5 model SST data

- Filtering regarding variable
- Geometric adjustments
- Spatial and temporal subsetting
- Quality assessment of model data by means of satellite-observed SST data using plug-in/API (user-determined validation method), discarding of models undercutting certain values
- Application of best models for trend analysis (removal of seasonal cycles)
- Calculation of SST anomaly/increase values for several time steps compared with reference data (ensemble mean and spread/uncertainties), construct probability density functions, examination of differing results by ANOVA
- Visualisation
- Data export

5.21 Investigation of Relationships between Ice Sheet ECV Parameters

User Types

- Earth system science community

Problem Definition A scientist wants to gain insight into the relationship between the Ice Sheets CCI ECV parameters. At first, Surface Elevation Change (SEC), Ice Velocity (IV), and Gravitational Mass Balance (GMB) are compared. Afterwards, a basin-wise comparison of Surface Elevation Change averages and Gravimetry Mass Balance averages is conducted. And finally, vector and grid data are compared by co-plotting of IV and Calving Front Location (CFL) data. Additionally, it would be interesting to examine the relationships between sea ice, SST around Greenland, glacier melt respectively cloud cover and SEC/IV.

Required Toolbox Features

- Access to and ingestion of CCI Ie Sheets ECV data (SEC, IV, GMB)
- Re-gridding of all data to the SEC grid
- Display the data as different layers
- Calculation of the IV vector magnitude (per pixel) and display as a new layer
- Temporal interpolation of the SEC data to the GMB data times
- Calculation of the correlation coefficient (per pixel) between the SEC data and the GMB data for a given GMB measurement time, display as a new layer
- Access to and ingestion of a polygon shapefile corresponding to one of the GMB basins
- Filtering of the SEC values and the GMB values; discarding of the ones outside the GMB basin polygon
- Calculation of the average of the GMB and SEC values inside the basin polygon for each point in the time series
- Plotting of the averaged values in a time series plot, comparison with the provided GMB total basin values
- Access to and ingestion of the CCI Ice Sheets CFL time series; each element in the time series is a set of (lon/lat) line segments

- Plotting of the CFL line segments on top of the IV magnitude for different years

Optional

- Access to and ingestion of CCI ECV data (sea ice, SST, glaciers, clouds)
- Re-gridding of all data to the SEC grid
- Temporal and spatial subsetting
- Calculation of correlation coefficients
- Visualisation and export

5.22 Analysis of Equatorial Aerosol and Cloud Features using Hovmöller Diagrams

User Types

- Earth system science community

Problem Definition A scientist wants to analyze the relation of aerosols and clouds in the equatorial region (5° S–5° N) by means of Hovmöller diagrams displaying the equatorial mean value as portion of the mean value over all latitudes for cloud fraction and aerosol optical depth (y-axis e.g. months since 1980, x-axis longitudes e.g. 100° W–80° E).

Required Toolbox Features

- Access to and ingestion of ESA CCI Aerosol and Cloud data
- Geometric adjustments
- Temporal subsetting
- Calculation of requested anomaly values and side-by-side display of Hovmöller diagrams

CHAPTER 6

Operation Specifications

In this section, the non-trivial operations and data processors used in the ESA CCI Toolbox are specified. The term *operation* used here first refers to CCI Toolbox functions that will become part of the API as usual Python functions. Secondly and at the same time it refers to instances of such functions that are stored along with additional meta-data in the CCI Toolbox *operation registry* as described in *Operation Management*. The latter will be used to allow invocation of functions from the CCI Toolbox' command-line interface (CLI) and desktop application (GUI).

The intended readership of this chapter are software end users wishing to understand the details of the algorithms and methods used in the CCI Toolbox.

6.1 Subsetting and Selections

6.1.1 Subsetting and Selections (Category)

Operation Category

Operation Category name Subsetting and Selections

Description This Operation Category encompasses different operations for the Subsetting, selection and extraction of data.

Operations

Operation name Spatial Subsetting

Operation description creates spatial subsets

Operation name Temporal Subsetting

Operation description creates temporal subsets

6.1.2 Spatial Subsetting

Operation

Operation name Spatial Subsetting

Description This Operation provides the functionality to select data of a region of interest. All data outside will be discarded.

Utilised in ../uc_workflows/uc09_workflow

Options

name polygon subsetting

description spatial subsetting of data inside a polygon

name polygon subsetting by selection from a list of main regions

description analysis of propagation of uncertainties during geometric adjustment

name point subsetting

description selection of a spatial point to retrieve all temporal in formation of that point

Input data

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

name time (time, t)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since ...

name variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain variable

Output data

name subset of variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain variable for the chosen area of interest

Parameters

---

name lon, x (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

description longitudinal coordinate of point of interest

---

name lat, y (latitudinal position)

type floating point number

valid values [-90.; +90.]

description latitudinal coordinate of point of interest

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

more coordinates necessary for non-rectangular areas and 3D data

Example

```
#     Fortran example code for spatial subsetting/sub-setting
c     Spatial Subsetting
c-----Example region: n3.4
      x1=190.
      x2=240.
      y1=-5.
      y2=5.

      data_new=0.

      do t=1,nt
        do y=1,ny
          do x=1,nx
            if(lat(y).lt.y1.or.lat(y).gt.y2)then
              continue
            elseif(lon(x).lt.x1.or.lon(x).gt.x2)then
              continue
            else
              data_new(x,y,t)=data_old(x,y,t)
            endif
          enddo !x
        enddo !y
      enddo !t
c-------------------------------------------------------------
```

6.1.3 Spatial Subsetting

Operation

Operation name Spatial Subsetting

Description This Operation provides the functionality to select data of a region of interest. All data outside will be discarded.

Utilised in ../uc_workflows/uc09_workflow

Options

name polygon subsetting

description spatial subsetting of data inside a polygon

---

name polygon subsetting by selection from a list of main regions

description analysis of propagation of uncertainties during geometric adjustment

---

name point subsetting

description selection of a spatial point to retrieve all temporal in formation of that point

---

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name time (time, t)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

---

name variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain variable

Output data

---

name subset of variable

type floating point number

range [-infinity; +infinity]

dimensionality vector or cube or 4D

description values of a certain variable for the chosen area of interest

---

Parameters

---

name lon, x (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

description longitudinal coordinate of point of interest

---

name lat, y (latitudinal position)

type floating point number

valid values [-90.; +90.]

description latitudinal coordinate of point of interest

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

more coordinates necessary for non-rectangular areas and 3D data

---

Example

```
#     Fortran example code for spatial subsetting/sub-setting
c     Spatial Subsetting
c-----Example region: n3.4
      x1=190.
      x2=240.
      y1=-5.
      y2=5.

      data_new=0.

      do t=1,nt
        do y=1,ny
          do x=1,nx
            if(lat(y).lt.y1.or.lat(y).gt.y2)then
              continue
            elseif(lon(x).lt.x1.or.lon(x).gt.x2)then
              continue
            else
              data_new(x,y,t)=data_old(x,y,t)
            endif
          enddo !x
        enddo !y
      enddo !t
c-------------------------------------------------------------
```

6.2 Visualisation

6.2.1 Visualisation (Category)

Operation Category

Operation Category name Visualisation

Description This Operation Category encompasses different operations for the visualisation of data.

Operations

Operation name Table

Operation description displays a table

Operation name Time Series Plot

Operation description plots time series (point data or spatial mean of areal data)

Operation name Plot

Operation description plots a plot

Operation name Map

Operation description plots a map (data of one time step or temporal mean)

Operation name Animated Map

Operation description plots an animated map (data of different time steps)

6.2.2 Time Series Plot

Operation

Operation name Time Series Plot

Algorithm reference Wikpedia entry on time series (visualization)

Description This operation produces and displays one or multiple time series plots based on point data or the spatial mean of areal data.

Utilised in ../uc_workflows/uc09_workflow

Options

---

name plot anomalies

description plots anomalies instead of absolute values

settings reference period (or region) for anomaly calculation

---

name multiple datasets

description plots multiple time series on the same axes

---

name multiple datasets, lagged

description plots multiple time series on the same axes in a lagged manner

---

name plot settings

description settings for the plot

settings legend, colours, symbols

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (time, t)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name time series plot

type plot

description displays a time series plot (see *Options*)

Parameters

---

name x-axis annotation/label

type character

valid values all

default value probability, time, name of variable, . . . (depends on type of plot)

description label for x-axis

---

name y-axis annotation/label

type character

valid values all

default value name of variable (depends on type of plot)

description label for y-axis

---

name heading annotation/label

type character

valid values all

default value name of variable (depends on type of plot)

description text for image heading

6.2.3 Animated Map

Operation

Operation name Animated Map

Algorithm reference *Wikipedia entry on animated mapping <https://en.wikipedia.org/wiki/Animated_mapping>*

Description This operation produces and displays one or multiple animated map showing the data of different time steps.

Utilised in ../uc_workflows/uc09_workflow

Options

---

name plot anomalies

description plots anomalies instead of absolute values

settings reference period (or region) for anomaly calculation

---

name multiple datasets

description plots multiple animated maps (data of different time steps) side by side or as transparent layers

---

name map settings

description settings for the map

settings legend, land contours, north arrow, grid, ...

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (time ,t)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

name animated map(s)

type animated map

description displays one (mutliple) animated map(s side by side) (see *Options*)

Parameters

name x-axis annotation/label

type character

valid values all

default value probability, time, name of variable, . . . (depends on type of plot)

description label for x-axis

name y-axis annotation/label

type character

valid values all

default value name of variable (depends on type of plot)

description label for y-axis

name heading annotation/label

type character

valid values all

default value name of variable (depends on type of plot)

description text for image heading

6.2.4 Map

Operation

Operation name Map

Description This Operation serves for plotting of maps (data of one time step or temporal mean).

Utilised in ../uc_workflows/uc09_workflow, ../uc_workflows/uc06_workflow

Options

---

name plot anomalies

description plots anomalies instead of absolute values

settings reference period (or region) for anomaly calculation

---

name multiple datasets

description plots multiple animated maps (data of different time steps) side by side or as transparent layers

---

name map settings

description settings for the map

settings legend, land contours, north arrow, grid, ...

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name map

type map

description displays one (multiple) map(s side by side) (see *Options*)

Parameters

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name x-axis annotation/label

type character

valid values all

default value probability, time, name of variable, ... (depends on type of plot)

description label for x-axis

---

name y-axis annotation/label

type character

valid values all

default value name of variable (depends on type of plot)

description label for y-axis

---

name heading annotation/label

type character

valid values all

default value name of variable (depends on type of plot)

description text for image heading

6.2.5 Table

Operation

Operation name Table

Description This Operations serves for displaying of tables.

Utilised in ../uc_workflows/uc09_workflow, ../uc_workflows/uc06_workflow

---

Options

---

name table settings

description settings for the table

settings row-wise, column-wise, header, ...

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name table

type table

description displays a table (see *Options*)

Parameters

---

name start date

type *double?*

valid values *[1; infinity]*

default value first time step defined by input data

description first step of time period to be employed

---

name end date

type *double?*

valid values *[1; infinity]*

default value last time step defined by input data

description last step of time period to be employed

---

name lon, x (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value

-

description longitudinal coordinate of point of interest

---

name lat, y (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value

-

description latitudinal coordinate of point of interest

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

6.3 Geometric Adjustments

6.3.1 Geometric Adjustments (Category)

Operation Category

Operation Category name Geometric Adjustments

Description This Operation Category embraces Operations for different kinds of geometric adjustments like grid adaption, geospatial gap filling or matchup dataset generation.

Operations

Operation name Co-Registration

Operation description interpolates spatial data of one dataset (slave) onto the coordinate system of another dataset (master)

Operation name Reprojection

Operation description transfers data onto another coordinate system

---

Operation name Resampling

Operation description modifies the spatial resolution of a dataset (interpolation, extrapolation)

Operation name Geospatial Gap Filling

Operation description fills spatial gaps

Operation name Matchup Dataset Generation

Operation description finds matching points between a dataset of point data and a dataset of areal data

6.3.2 Co-Registration

Operation

Operation name Co-Registration

Algorithm reference *Wikipedia entry on image registration <https://en.wikipedia.org/wiki/Image_registration>*

Description This Operation interpolates spatial data of one dataset (slave) onto the coordinate system of another dataset (master).

Utilised in ../uc_workflows/uc09_workflow

Options

---

name master-slave setting

description which dataset tu use as master, which as slave

---

name interpolation method

description method for reprojecting the data

items nearest neighbor (primarily for thematic maps), bilinear, cubic convolution, spline

---

name propagation of uncertainties

description analysis of propagation of uncertainties during geometric adjustment

---

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain variable

Output data

---

name adjusted longitude (lon', x')

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description new grid information on longitudes

---

name adjusted latitude (lat', y')

type floating point number

range [-90.; +90.]

dimensionality vector

description new grid information on latitudes

---

name adjusted height (z')

type floating point number

range [-infinity; +infinity]

dimensionality vector

description new grid information on height/depth

---

name adjusted variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description new values of a certain variable

Parameters

name original coordinate system

description definition of original coordiate system

name adjusted coordinate system

description definition of requested coordiate system

6.3.3 Geospatial Gap Filling

Operation

Operation name Geospatial Gap Filling

Description This Operation served for the filling of spatial gaps.

Applicable use cases UC11

Options

name geospatial gap filling method

description method to fill spatial gaps

items linear, bi-cubic, . . .

Input data

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain variable

Output data

---

name adjusted variable data

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description new values of a certain variable

Parameters

---

name nx

type integer

valid values [1; infinity]

default value number of longitudes in dataset

description original number of longitudes

---

name ny

type integer

valid values [1; infinity]

default value number of latituted in dataset

description original number of latitudes

---

name nx'

type integer

valid values [1; infinity]

default value

-

description adjusted number of longitudes

---

name ny'

type integer

valid values [1; infinity]

default value

-

description adjusted number of longitudes

---

name size of sliding window

type integer

valid values [1; infinity]

default value 3

description side length of the sliding window used for interpolation and/or gap filling (e.g. 3x3, 9x9). For some tasks solely odd numbers are applicable.*???*

---

name original coordinate system

description definition of original coordiate system

---

name adjusted coordinate system

description definition of requested coordiate system

6.3.4 Reprojection

Operation

Operation name Reprojection

Description This Operation converts spatial data from one coordinate system to another.

Utilised in ../uc_workflows/uc09_workflow

Options

---

name coordinate system

description requested coordinate system

items UTM, geographic, regular lat-lon, polar centred, sinusoidal, tripolar, . . .

---

name ellipsoid

description requested ellipsoid

items WGS84, GRS80, Bessel, Clarke

---

name reprojection method

description method for reprojecting the data

items nearest neighbor (primarily for thematic maps), bilinear, cubic convolution

---

name propagation of uncertainties

description analysis of propagation of uncertainties during geometric adjustment

---

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain variable

Output data

---

name adjusted longitude (lon', x')

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description new grid information on longitudes

---

name adjusted latitude (lat', y')

type floating point number

range [-90.; +90.]

dimensionality vector

description new grid information on latitudes

---

name adjusted height (z')

type floating point number

range [-infinity; +infinity]

dimensionality vector

description new grid information on height/depth

---

name adjusted variable

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description new values of a certain variable

Parameters

---

name nx

type integer

valid values [1; infinity]

default value number of longitudes in dataset

description original number of longitudes

---

name ny

type integer

valid values [1; infinity]

default value number of latitudes in dataset

description original number of latitudes

---

name nz

type integer

valid values [1; infinity]

default value number of altitude levels in dataset

description original number of altitude levels

---

name nx'

type integer

valid values [1; infinity]

default value

-

description adjusted number of longitudes

---

name ny'

type integer

valid values [1; infinity]

default value

-

description adjusted number of latitudes

---

name nz'

type integer

valid values [1; infinity]

default value

-

description adjusted number of altitude levels

---

name size of sliding window

type integer

valid values [1; infinity], odd numbers

default value 3

description side length of the sliding window used for interpolation (e.g. 3x3, 9x9)

---

name original coordinate system

description definition of original coordiate system

---

name adjusted coordinate system

description definition of requested coordiate system

6.4 Data Inter-Comparison

6.4.1 Data Inter-Comparison (Category)

Operation Category

Operation Category name Data Inter-Comparison

Description This Operation Category includes various Operations for bivariate statistical analysis.

Operation Subcategories

Operation Subcategory name Contingency Table

Operation Subcategory description creation of a contingency table showing the frequency distribution of variable1 and variable2, derivation of related quantitites

Operations Contingency Table, Marginal Probabilities, Conditional Relative Probabilities, Test on Independency

---

Operation Subcategory name Location Parameters

Operation Subcategory description calculation of one single value for data description

Operations Arithmetic Mean Center, Median Center

Operation Subcategory name Dispersion Parameters

Operation Subcategory description calculation of a measure for the dispersion of the data inside the sample

Operations Standard Distance

Operation Subcategory name Correlation Analysis

Operation Subcategory description calculation of measures for the strength and direction of the connection between variable1 and variable2.

Operations Standardized Contingency Coefficient, Rank Correlation Coefficient (Spearman), Product-Moment Correlation Coefficient (Pearson)

Operation Subcategory name Regression Analysis

Operation Subcategory description calculation of coefficients of a function which serves as an approximation of the connection between variable1 and variable2; one of both variables is assumed to be regressor respectively predictor (independent variable), the other as regressand (dependent variable)

Operations Linear Regression Analysis, Determination Coefficient, Non-Linear Regression Analysis

6.4.2 Correlation Analysis (Category)

Operation Subcategory name Correlation Analysis

Description This Operation Subcategory comprises different types of Correlation Analysis for the calculation of measures for the strength and direction of the connection between two variables which are suitable for different scales of data.

Operations

Operation name Standardized Contingency

Operation description performs a correlation analysis of nominally scaled data

---

Operation name Rank Correlation (Spearman)

Operation description performs a correlation analysis of ordinally scaled data

---

Operation name Product-Moment Correlation (Pearson)

Operation description performs a correlation analysis for metrically scaled data

6.4.3 Product-Moment Correlation (Pearson)

Operation

Operation name Product-Moment Correlation (Pearson)

Algorithm reference Wikipedia entry on Pearson product-moment correlation coefficient

Description This Operation performs a correlation analysis for metrically scaled data (assumption: normal distribution).

Utilised in ../../uc_workflows/uc09_workflow

Options

---

name temporal correlation

description performs a correlation analysis regarding temporally variable values

items one grid cell, cell-by-cell, spatial mean

---

name spatial correlation

description performs a correlation analysis regarding spatially variable values

items one point in time, time-by-time, temporal mean

---

name scatter-plot

description displays a scatter-plot showing corresponding variable values (not for time-by-time and pixel-by-pixel analysis)

---

name time series plot

description plots results for spatial time-by-time correlation

---

name map

description produces and displays a map showing cell-by-cell correlations

---

name table

description produces a table listing pixel-by-pixel respectively time-by-time correlation coefficients

---

name t test

description performs a t test to assess the significance level of the results

---

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable1

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain geophysical quantity

---

name variable2

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of a certain geophysical quantity

---

name time (time, t)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name product-moment correlation coefficient (Pearson)

type floating point number

range [-1.; +1.]

dimensionality scalar

description for correlation analysis for metrically scaled data

---

name signficance

type boolean

range {0,1}

dimensionality scalar

description significant or non-significant

alternatively

name level of signficance

type floating point number

range [0; +infinity]

dimensionality scalar

description significance level of correlation

---

name scatter plot

description displays a plot (see *Options*)

---

name time series plot

description displays a time series plot (see *Options*)

---

name map

description displays a map (see *Options*)

---

name table

description displays a table (see *Options*)

Parameters

---

name level of significance

type floating point number

valid values [0; 1]

default value 0.95

description level of significance for t test, determines t value to be compared with test value

---

for plot settings, the procedure is forwarded to the Visualisation Operation

---

Example

```
# Fortran subroutine for product moment correlation analysis (includes mean value␣
↪function)

c-----subroutine "correlation"
c.....calculation of
c.....a) product-moment corellation coefficient "cc" between x(t) and y(t), t=[1,nt]
c.....b) test-value "test" for t-test
      subroutine s_correlation(nt,x,y,cc,test) !Zeit
      implicit none
      integer nt,t
      real x(nt),dummy,dummy2,dummy3,y(nt),cc,test,f_mw

      dummy=0.
      dummy2=0.
      dummy3=0.
      do t=1,nt
        dummy=dummy+((x(t)-f_mw(n,x))*(y(t)-f_mw(n,y)))
        dummy2=dummy2+((x(t)-f_mw(n,x))**2)
        dummy3=dummy3+((y(t)-f_mw(n,y))**2)
      enddo !ja
      cc=(dummy)/sqrt(dummy2*dummy3)
      test=cc*sqrt((n-2)/(1-(cc**2)))

      return
      end

c-----function "mean value"
c.....calculation of mean value f_mw(nt,x) of vairable x with a sample size nt
      real function f_mw(nt,x)
      implicit none
      integer nt,t
      real x(nt)

       f_mw=0.
      do t=1,nt
        f_mw=f_mw+x(t)
      enddo
      f_mw=f_mw/float(nt)

      return
      end
```

6.5 Calculations

6.5.1 Calculations (Category)

Operation Category

Operation Category name Calculations

Description This Operation Category encompasses different operations for the ... of data.

Operations

Operation name Seasonal Values

Operation description calculates seasonal values

Operation name Arithmetics

Operation description for simple arithmetic data manipulation (log transformation, adding/subtracting/multiplying/dividing constants etc.)

Operation name Index Calculation

Operation description calculation of (pre-defined) indices involving spatial and temporal averaging, anomalies, standardization, filtering, etc

---

6.5.2 Seasonal Values

Operation

Operation name Seasonal Values

Description This Operation serves for the calculation of seasonal values.

Utilised in ../uc_workflows/uc06_workflow

Options

---

name season of interest

description definition of season of interest

settings starting point, terminating point

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name seasonal values

type same as input data

description seasonal values of input data for user-defined season

---

Parameters

---

name time1, tim1

type *character*

valid values *{01-01, . . . , 12-31}*

default value January 1st

description starting point of season of interest

---

name time2, tim2

type *character*

valid values *{01-01, . . . , 12-31}*

default value December 31st

description terminal point of season of interest

6.5.3 Arithmetics

Operation

Operation name Arithmetics

Description This Operation serves for simple arithmetic data manipulation (log transformation, adding/subtracting/multiplying/dividing constants etc.) by defining equations.

Utilised in ../uc_workflows/uc06_workflow

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

name result

type same as input data

description result of arithmetic manipulation of input data

6.5.4 Index Calculation

Operation

Operation name Index Calculation

Description This Operation serves for calculation of (pre-defined) indices involving spatial and temporal averaging, anomalies, standardization, filtering etc.

Utilised in ../uc_workflows/uc06_workflow

Options

name Niño3.4 index

description five month running mean of anomalies of monthly means of SST in Niño3.4 region (120°W-170°W, 5°S- 5°N), see UCAR webpage on El Niño indices

settings time series calculation, Boolean El Niño (threshold +0.4 deg C), Boolean La Niña (threshold -0.4 deg C)

Input data

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name index timeseries (values or Boolean)

type floating point or Boolean

description timeseries of values or Boolean results of index calculation

---

6.6 Complex Computations

6.6.1 Complex Computations (Category)

Operation Category

Operation Category name

Description This Operation Category encompasses different operations for complex data analysis computations.

Operations

Operation name EOF Analysis

Operation description Empricial Orthogonal Function (EOF) Analysis, also known as Principal Component Analysis (PCA). For data analysis regarding spatial patterns/modes.

6.6.2 EOF Analysis

Operation

Operation name EOF Analysis

Algorithm reference *Wikipedia entry on Principal Component Analysis <https://en.wikipedia.org/wiki/Principal_component_analysis>*, *Blog entry on step by step PCA implementation in Python <http://sebastianraschka.com/Articles/2014_pca_step_by_step.html>*,

Description This Operations serves for the application of Empricial Orthogonal Function (EOF) Analysis, also known as Principal Component Analysis (PCA), for data analysis regarding spatial patterns/modes. EOF Analysis implies the removal of redundancy.

Utilised in ../uc_workflows/uc06_workflow

Options

---

name rotated

description decide if EOF analysis should be rotatated

settings no rotation, varimax, . . .

---

name matrix

description decide to use correlation or covariance matrix

settings correlation matrix or covariance matrix

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name principal components (PCs)

type floating point number

range [-infinity.; +infinity]

dimensionality vector

description temporal evolution of variance belonging to spatial pattern, number of

---

name empirical orthogonal functions (EOFs)

type floating point number

range [-infinity.; +infinity]

dimensionality array

description also named eigenvectors; tendency and strength of dominant spatial pattern of variance. All eigenvectors are orthogonal to one another.

---

name eigenvalues

type floating point number

range [0; 1] for correlation matrix, [0; +infinity] for covariance matrix

dimensionality scalar

description ith eigenvalue constitutes measure for the portion of variance explained by the ith PC/EOF

Parameters

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

6.7 Univariate Descriptive Statistics

6.7.1 Univariate Descriptive Statistics (Category)

Operation Category

Operation Category name Univariate Descriptive Statistics

Description This Operation Category encompasses different operations for the internal analysis and manipulation of a data product.

Operation Subcategories

Operation Subcategory name Location Parameters

Operation Subcategory description calculations of measures to describe the location of elements with respect to the sample.

Operations Arithmetic Mean (temporal, spatial; weigthing option), Percentiles and Median, Modus, Geometric Mean

---

Operation Subcategory name Comparison

Operation Subcategory description calculation of measures to internally compare a dataset (temporal, spatial, . . .)

Operations Relative Values, Anomalies, Standardization, Cumulative Changes, Hovmöller Analysis

---

Operation Subcategory name Filtering

Operation Subcategory description calculations to manipulate the data in a way to highlight or remove specific features

Operations Detection of Outliers, Filtering (High Pass, Low Pass, Band Pass), Removal of Seasonal Cycles

6.7.2 Comparison (Subcategory)

Operation Subcategory

Operation Subcategory name Comparison

Description This Operation Subcategory encompasses different operations for the internal comparison of data.

Operations

Operation name Anomalies

Operation description calculates differences compared to a reference (temporal, spatial mean; reference period, reference region)

Operation name Long-term average

Operation description calculates a long term average which can be used as a reference for calculating anomalies

6.7.3 Anomalies

Operation

Operation name Anomalies

Description This Operation serves for the calculation of differences compared to a reference.

Utilised in ../../uc_workflows/uc09_workflow, ../../uc_workflows/uc06_workflow

---

Options

---

name temporal

description calculate anomalies compared to the temporal mean of a specific reference period

settings reference period

---

name spatial

description calculate anomalies compared to the spatial mean of a specific reference region

settings reference region

---

name internal reference

description calculate anomalies compared to the mean of a specfic region/time of the input data itself.

settings reference region, reference period

---

name external reference

description calculate anomalies compared to the mean of a specfic region/time of external reference data.

settings reference region, reference period, reference data

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since ...

Output data

---

name anomalies

type array

description input data transformed to anomalies (same dimensions as input data)

Parameters

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

---

name time1, tim1

type integer or double

valid values [0; +infinity]

default value start point of input data

description starting point of reference period

---

name time2, tim2

type integer or double

valid values [0; +infinity]

default value terminal point of input data

description terminal point of reference period

6.7.4 Long-term average

Operation

Operation name Long-term average

Description This Operation serves for the calculation of long-term averages as reference.

Utilised in ../../uc_workflows/uc06_workflow

Options

---

name preserve saisonality

description calculate long-term mean for every timestep inside a year (month, day, . . .)

settings reference period

---

name one value

description calculate one long-term mean without preserving saisonality

settings reference period

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

Output data

---

name long-term average

type floating point number

dimensionality one value or vector

description input data transformed to long-term average

Parameters

---

name time1, tim1

type integer or double

valid values [0; +infinity]

default value start point of input period

description starting point of reference period

---

name time2, tim2

type integer or double

valid values [0; +infinity]

default value terminal point of input period

description terminal point of reference period

Example

```
# ny number of years
# variable: var(year, month)

##############################
# with saisonality
do month=1,12
        longtermmean(month)=mean(var(year, month), year=1,ny)
enddo

#anomaly
var(year, month)=var(year,month)-longtermmean(month)

##############################
# without seasonality

longtermmean=mean(var)

#anomaly
var(year, month)=var(year,month)-longtermmean
```

6.7.5 Location Parameters (Subcategory)

Operation Subcategory

Operation Subcategory name Location Parameters

Description This Operation Subcategory encompasses different operations for the calculations of measures to describe the location of elements with respect to the sample.

Operations

Operation name Arithmetic Mean

Operation description calculates the artihmetic mean

Options temporal, spatial, weighted

6.7.6 Aritmetic Mean

Operation

Operation name Arithmetic Mean

Algorithm name *XXX*

Algorithm reference *XXX*

Description This operation serves for the calculation of arithmetic means.

Utilised in ../../uc_workflows/uc09_workflow, ../../uc_workflows/uc06_workflow

---

Options

---

name temporal

description for the calculation of temporal means

---

name spatial

description for the calculation of spatial means

---

name spatio-temporal

description for the calculation of spatiotemporal means

---

name weighting

description for the calculation of weighted means

settings weighting factors

Input data

---

name longitude (lon, x)

type floating point number

range [-180.; +180.] respectively [0.; 360.]

dimensionality vector

description grid information on longitudes

---

name latitude (lat, y)

type floating point number

range [-90.; +90.]

dimensionality vector

description grid information on latitudes

---

name height (z)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description grid information on height/depth

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality cube or 4D

description values of (a) certain variable(s)

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

---

name weighting factors

type floating point

range [0; +infinity]

dimensionality vector or array

description weighting factors, same dimensions as input data

Output data

---

name arithmetic mean

type floating point

description arithmetic mean of the input data (details see *Options*)

6.7.7 Filtering (Subcategory)

Operation Subcategory

Operation Subcategory name Filtering

Description This Operation Subcategory encompasses different operations for the filtering of data.

Operations

Operation name Detection of Outliers

Operation description Detect outliers within a sample.

Options threshold-limitation, percentile-limitation

6.7.8 Detection of Outliers

Operation

Operation name Detection of Outliers

Algorithm name *XXX*

Algorithm reference *XXX*

Description This Operation enables the detection of outliers within a sample.

Options

---

name percentile-approach

description identify valid values inside the range of two limiting percentiles

settings ** to be defined later **

---

name threshold-approach

description identify valid values within a given range determined by two threshold values

settings ** to be defined later **

Input data

---

name time (steps)

type integer or double

range [0; +infinity]

dimensionality vector

description days/months since . . .

---

name variable(s)

type floating point number

range [-infinity; +infinity]

dimensionality vector

description values of (a) certain variable(s)

Output data

---

name cleaned sample

type floating point number

range [-infinity; +infinity]

dimensionality vector

description clean input after outliers have been removed

Parameters

---

name lon1, x1 (longitudinal position)

type floating point number

valid values [-180.; +180.] respectively [0.; 360.]

default value minimum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lon2, x2 (longitudinal position)

type floating point number

valid values [-180.; +180.] resp. [0.; 360.]

default value maximum longitude of input data

description longitudinal coordinate limiting rectangular area of interest

---

name lat1, y1 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value minimum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

name lat2, y2 (latitudinal position)

type floating point number

valid values [-90.; +90.]

default value maximum latitude of input data

description latitudinal coordinate limiting rectangular area of interest

Example

```
'''The following program is an example for the 'Detection of Outliers'. The suggested␣
↪method is a detection of outliers
 based on percentiles or threshold-limitation.

 Step 1:
A random dataset with a length of 95 floats within the span of 15 and 25 is generated␣
↪randomly. Five outlier values are
added by hand.

Step 2:
Prompt:: Decide between the two approaches/methods.

Step 3:
Prompt:: Set limitations either a percentage [%] or a value embracing the␣
↪distribution.

Step 4:
Prompt:: Flag or drop the outliers. If falgged: column_stack a new column with 0/1. '1
↪' flags an outlier.

Step 5:
Implemt of an 'R-like' which()-statement.

Step 6: Exclude or flag the values.

Return-Object: 'new_sampl' based on the prior decisions.

#Comment: This method of detecting outliers is just one of many! UC2 is a perfect␣
↪example of a 'Detection o Outliers'
via two threshold-values giving a rigid limition for the span of values allowed. When␣
↪the data is assumed to be tempera-
tures in Celius measured during the summer. I.e. the User could save drop/flag all␣
↪values lower 15 and greater 25,
since the temperature in the given period is considered to vary in that range.

02.02.2017 Stephan Herzog
'''

#import modules
import numpy as np

## - TEST DATA - ##
```

(continues on next page)

(continued from previous page)

```
#Generate 95 random values within 15 and 25; pass it to 'vec1'
sampl = np.random.uniform(low=15.0,high=25.0,size=95)
sampl = np.append(sampl,[-3.141,42,1337,-273.15,21122012])
np.random.shuffle(sampl)


######BEGIN: VOR DEM PROMPT DIE ABFRAGE EINBAUEN OB PERCENTIL_METHODE ODER␣
↪SCHWELLWERT!!!!
logical_prompt = raw_input("Please decide between the methods for a detection of␣
↪outliers: Press (1) for a percentile-"
                                                    "approach; Press (2) for a␣
↪threshold-approach.")

## - Calc. of percentiles - ##
if (logical_prompt == '1') :
        prompt1lower = raw_input("Please enter the lower limit for the percentile: ")␣
↪  ##Suggestion: 2.5
        prompt2upper = raw_input("Please enter the upper limit for the percentile: ")␣
↪  ##Suggestion: 97.5

        p_lower = np.percentile(sampl, float(prompt1lower))     ##key aspect
        p_upper = np.percentile(sampl, float(prompt2upper))     ##key aspect

## - Prompt for threshold - ##
if (logical_prompt == '2') :
        p_lower = raw_input("Please enter the lower limit for the threshold: ")
        p_upper = raw_input("Please enter the upper limit for the threshold: ")

        p_lower = float(p_lower)
        p_upper = float(p_upper)

## - Prompt for flag or drop - ##
logical = raw_input("Should the outliers be flagged? (Y/N)")

## - Identfiy values within limits - ##
which = lambda lst:list(np.where(lst)[0])       ##key aspect

lst = map(lambda x:(x<p_lower or x>p_upper), sampl)

print(which(lst))
## - Flag or Drop Outliers - ##
if ( logical == 'Y') :
        flag = np.repeat(0,len(sampl))
        flag[which(lst)] = 1
        new_sampl = np.column_stack((sampl,flag))
        print(new_sampl.shape)
        print(new_sampl[which(lst),:])
else:
        new_sampl = np.delete(sampl,which(lst))
        print(new_sampl.shape)

## - Write to Output - ## e.g. .csv or other
```

CHAPTER 7

Architecture

This chapter describes the internal, technical design of the CCI Toolbox that has been developed on the basis of the CCI Toolbox User Requirements Document (URD), the climate data exploitation *Use Cases* defined in the URD, as well as the abstract *Operation Specifications* that have been derived from both.

This architecture description tries to reflect the current software design of the CCI Toolbox and should provide the big picture of the software to the development team and should help other programmers getting an overview.

Please note that this architecture description does not necessarily reflect the CCI Toolbox application programming interface (API). The actual public API comprises a relatively stable subset of the components, types, interfaces, and variables describes here and is described in chapter *API Reference*.

7.1 Overview

The CCI Toolbox at its basis is a Python package `cate` which provides the a command-line interface (CLI), application programming interface (API), and a web API interface (WebAPI), and also implements all required climate data visualisation, processing, and analysis functions. It defines a common climate data model and provides a common framework to register, lookup and invoke operations and workflows on data represented in the common data model.

The CCI Toolbox graphical user interface, the GUI, is based on web technologies, i.e. JavaScript and HTML-5, and communicates with the Python core via its WebAPI. The GUI is designed as a native desktop application (uses Electron technology for the desktop operating system integration). It will us a Python (RESTful) web server running on the user's computer and providing the CCI Toolbox' WebAPI service to the GUI. This design allows for later extensions towards a web application with possibly multiple remote WebAPI services.

The ESA CCI Open Data Portal is the central climate data provider for the CCI Toolbox. It provides time series of essential climate variables (ECVs) in various spatial and temporal resolutions in netCDF and Shapefile format. At the time of writing (June 2016), the only operational data access service is via FTP. However, the CCI Open Data Portal will soon offer also data access via a dedicated THREDDS server and will support *OPEeNDAP* and *OGC WCS* services.

The following Fig. 7.1 shows the CCI Toolbox GUI, CCI Toolbox Python core, and the CCI Open Data Portal.

Note that although the CCI Toolbox GUI and Python core are shown in Fig. 7.1 as separate nodes, they are combined in one software installation on the user's computer.

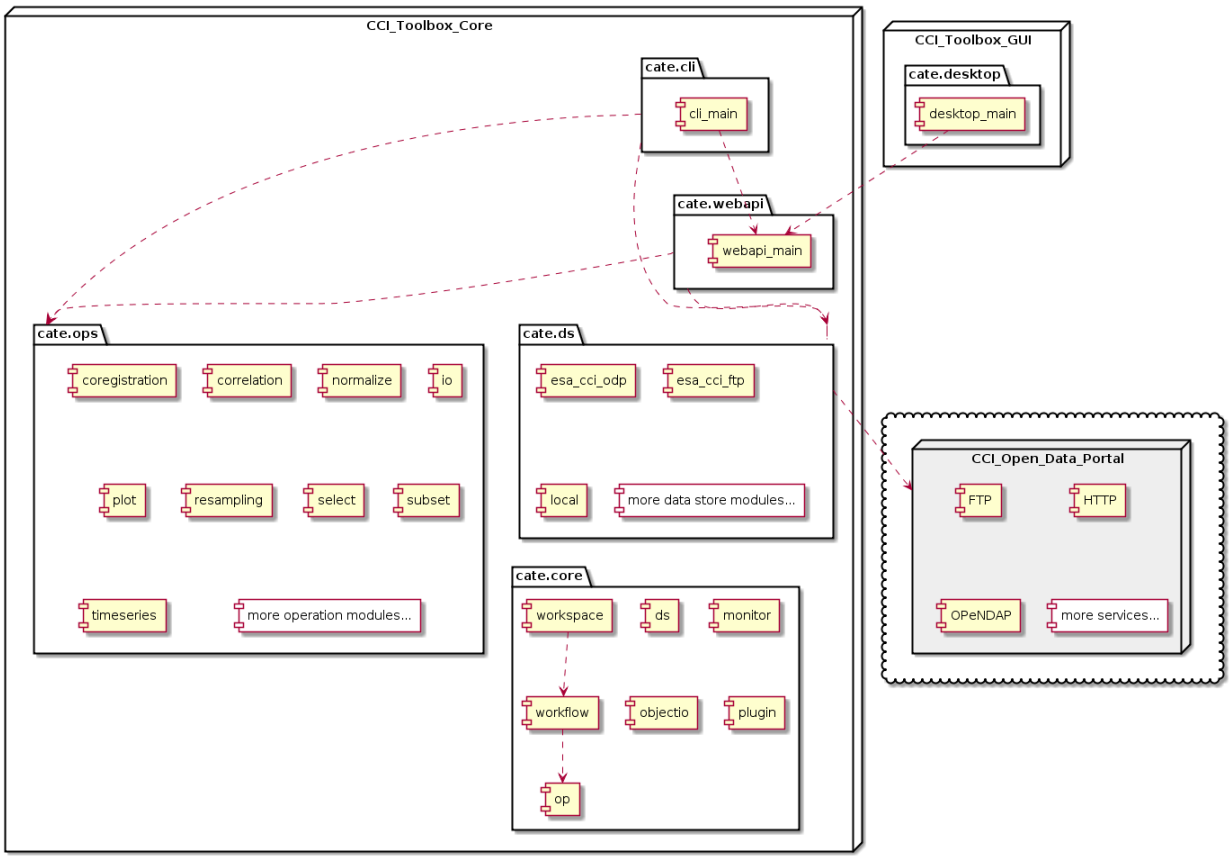


Fig. 7.1: CCI Toolbox GUI, CCI Toolbox `cate` package, and the CCI Open Data Portal.

The CCI Toolbox Python package comprises several sub-packages of which are described in the following four sections.

7.1.1 Package `cate.core`

The `cate.core` Python package is most import part of the CCI Toolbox architecture. It provides a common framework for climate data I/O and processing. Although designed for climate tooling and use with climate data the framework and API is more or less application-independent.

The `cate.core` package

- defines the CCI Toolbox' common data model
- provides the means to read climate data and represent it in the common data model
- provides the means to process / transform data in the common data model
- to write data from the common data model to some external representation

As a framework, `cate.core` allows plugins to extend the CCI Toolbox capabilities. The most interesting extension points are

- climate data stores that will be added to the global data store registry
- climate data visualisation, processing, analysis operations that will be added to the global operations registry

The `cate.core` packages comprises the essential modules which described in more detail in the following subsections:

- module `ds` - *Data Stores and Data Sources*
- module `op` - *Operation Management*
- module `workflow` - *Workflow Management*
- module `objectio` - *Object Input/Output*
- module `plugin` - *Plugin Concept*

7.1.2 Package `cate.ds`

The Python package `cate.ds` contains specific climate data stores (=ds). Every module in this package is dedicated to a specific data store.

- The `esa_cci_odp` module provides the data store that allows opening datasets provided by the ESA CCI Open Data Portal (ODP). More specifically, it provides data for the `esacii` entry in the ESGF data service.
- The `esa_cci_ftp` module provides the data store that allows opening datasets provided by the FTP service of the ESA CCI Open Data Portal. This data store is now deprecated in favour of the ESGF service.

The package `cate.ds` is a *plugin* package. The modules in `cate.ds` are activated during installation and their data sources are registered once the module is imported. In fact, no module in package `cate.core` has any knowledge about the package `cate.ds` and users never deal with its modules directly. Instead, all registered data stores are accessible through the `cate.core.ds.DATA_STORE_REGISTRY` singleton.

7.1.3 Package `cate.ops`

The Python package `cate.ops` contains (climate-)specific visualisation, processing and analysis functions. Every module in this package is dedicated to a specific operation implementation. For example the `timeseries` module provides an operation that can be used to extract time series from datasets. Section *Operation Management* describes

the registration, lookup, and invocation of operations, section *Workflow Management* describes how an operation can become part of a workflow.

The chapter *Operation Specifications* provides abstract descriptions of the individual operations in this package.

Similar to `cate.ds`, the package `cate.ops` is a *plugin* package, only loaded if requested, and no module in package `cate.core` has any knowledge about the package `cate.ops`.

7.1.4 Package `cate.cli`

The package `cate.cli` comprises a `main` module, which implements the CCI Toolbox' command-line interface.

The command-line interface is described in section *Command-Line Interface*.

7.1.5 Package `cate.webapi`

The package `cate.webapi` implements the CCI Toolbox' *WebAPI* which implements a web service that allows using the CCI Toolbox Python API from the * Desktop GUI as well as * the interactive commands of the CLI.

7.1.6 Package `cate.util`

The `cate.util` package is fully application-independent and can be used stand-alone. Numerous, CCI Toolbox API functions take a `monitor` argument used for progress monitoring of mostly long-running tasks. The `cate.util.monitor` package defines the `Monitor` class.

- module `monitor` - *Task Monitoring*

7.1.7 Package `cate.conf`

The `cate.conf` package provides Cate's configuration API. The `cate.conf.defaults` module defines the default values for Cate's configuration parameters.

7.2 Common Data Model

The primary data source of the first releases of the CCI Toolbox are the data products delivered by the ESA CCI programme. Later in the project, the CCI Toolbox will also address other datasets.

The majority of the gridded ECV datasets from ESA CCI are in *netCDF-CF* format, which is a de-factor standard in climate science. The datasets of the Land Cover CCI are provided in *GeoTIFF* format and the Glaciers and Ice Sheets CCIs deliver their datasets in *ESRI Shapefile* format.

Ideally, the CCI Toolbox could combine the various datasets in a single *common data model* so that an API could be designed that allows a uniform and transparent for data access. This would also allow to make a maximum of operations work on both raster and vector data.

As this sounds reasonable at first, the team has decided not go for such a grand unification as the way how gridded raster data is processed is substantially different from how vector data is processed. To make the majority of data operations applicable to both data types, rasterisation (or vectorisation) would need to occur implicitly and would need to be controlled by explicit operation parameters.

Instead, the CCI Toolbox stays with the Unidata Common Data Model and CF Conventions for raster data, and the Simple Features Standard (ISO 19125) for vector data. This is achieved by reusing the data models and APIs of the popular, geo-spatial Python libraries.

7.2.1 Raster Data

For the representation of raster or gridded data, the CCI Toolbox relies on the xarray Python library. `xarray` builds on top of numpy, the fundamental package for scientific computing with Python, and pandas, the Python Data Analysis Library.

The central data structure in the CCI Toolbox is xarray.Dataset, which is an in-memory representation of the data model from the netCDF file format. Because of its generality for multi-dimensional arrays, it is also well-suited to represent the GeoTIFF and other raster and gridded data formats. The `xarray.Dataset` structure is composed of the following elements and follows the Unidata Common Data Model:

Variables are containers for the dataset's geo-physical quantities. They are named, multi-dimensional arrays of type xarray.DataArray which behave quite like numpy ndarrays. The dataset variables are accessible through the `data_vars` attribute, which is mapping from variable name to the multi-dimensional data arrays.

Coordinates To label the grid points contained in the variable arrays, *coordinates* are used. Coordinates are also xarray.DataArray instances and are accessible through the `coords` attribute, which is a mapping from coordinate names to the usually one-dimensional label arrays.

Dimensions All dimensions used by the variables and coordinates arrays are named and have a size. The mapping from dimension name to size is accessible through the `dims` attribute.

Attributes are used to hold metadata both for `xarray.Dataset` and `xarray.DataArray` instances. Attributes are accessed by the `attrs` attribute which is a mapping from attribute names to arbitrary values.

7.2.2 Vector Data

From version 1.0 on, the representation of vector data will be provided by utilising the GeoPandas Python library. Similar to xarray, also `GeoPandas` relies on pandas, the Python Data Analysis Library.

Once the CCI Toolbox supports vector data, it will provide a rasterisation operation in order to convert vector data into the raster data model, namely xarray.Dataset instances.

7.3 Data Stores and Data Sources

In the CCI Toolbox, a *data store* represents something that can be queries for climate *data sources*.

For example, the ESA CCI Open Data Portal currently (June 2016) provides climate data products for around 13 essential climate variables (ECVs). Each ECV comes in different spatial and temporal resolutions, may originate from various sensors and may be provided in various processing versions. A *data source* refers to such a unique ECV occurence.

The `cate.core.ds` module comprises the following abstract types:

The `DataStoreRegistry` manages the set of currently known data stores. The default data store registry is accessible via the variable `DATA_STORE_REGISTRY`. Plugins may register new data stores here. There will be at least one data store available which is by default the data store that mirrors parts of the FTP tree of CCI Open Data Portal on the user's computer.

The `DataStore.query()` allows for querying a data store for data sources given some optional constraints.

The actual data of a data source can be provided by calling the `DataSource.open_dataset()` method which provides instances of the `xarray.Dataset` type which has been introduced in the former section *Package cate.util*.

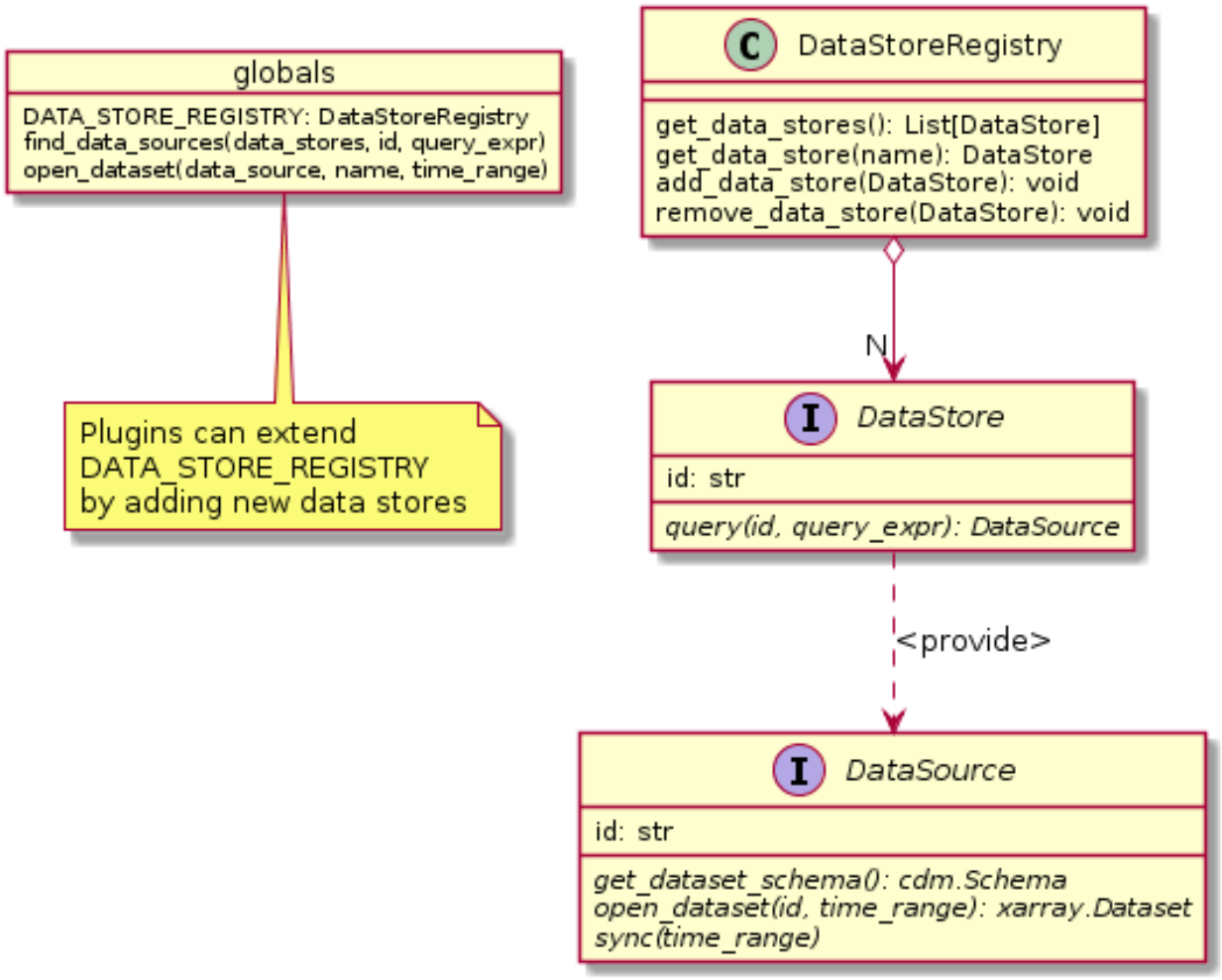


Fig. 7.2: DataStore and DataSource

The `DataSource.sync()` method is used to explicitly synchronise the remote content of a data store with locally cached data.

7.4 Operation Management

The CCI Toolbox `cate.core.op` module allows for the registration, lookup and controlled invocation of *operations*. Operations can be run from the CCI Toolbox command-line (see next section *Command-Line Interface*), may be referenced from within processing *workflows* (see next section *Workflow Management*), or may be invoked from from the WebAPI (see Fig. 7.1) as a result of a GUI request.

An operation is represented by the `Operation` type which comprises any Python callable (function, lambda expression, etc.) and some additional meta-information `OpMetaInfo` that describes the operation and allows for automatic input validation, input value conversion, monitoring. The `OpMetaInfo` object specifies an operation's signature in terms of its expected inputs and produced outputs.

The CCI Toolbox framework may invoke an operation with a `Monitor` object, if the operation supports it. The operation can report processing progress to the monitor or check the monitor if a user has requested to cancel the (long running) operation.

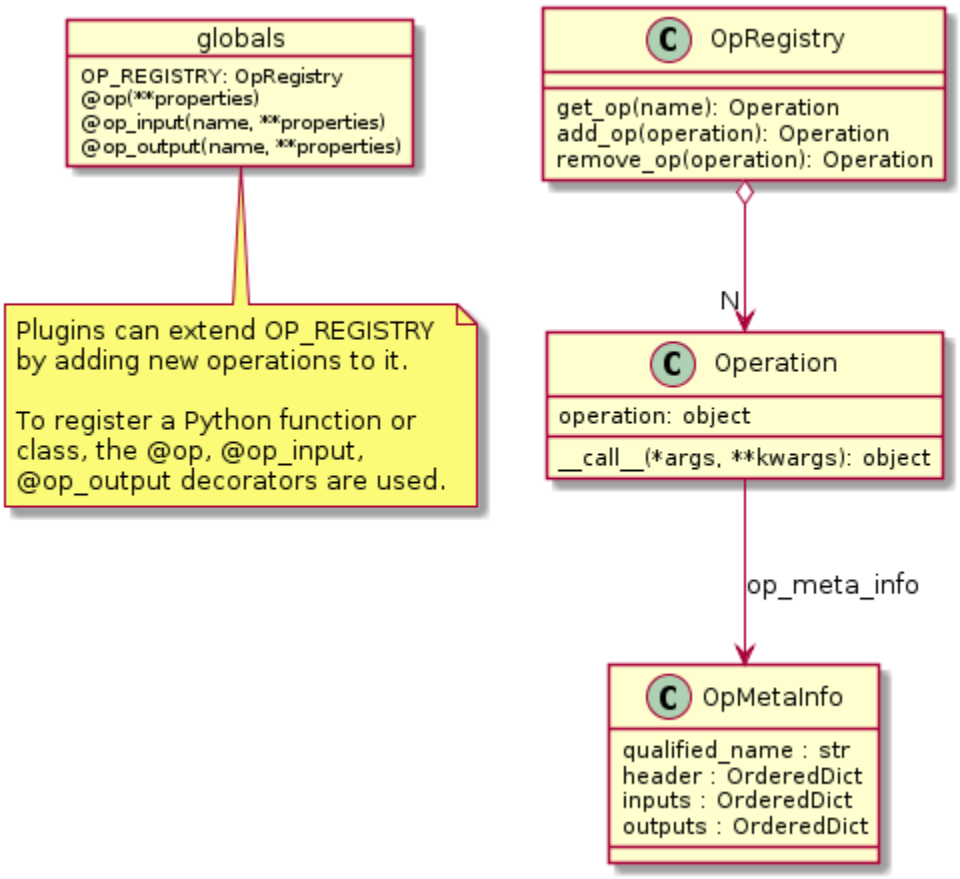


Fig. 7.3: OpRegistry, Operation, OpMetaInfo

Operations are registered in operation registries of type `OpRegistry`, the default operation registry is accessible via the global, read-only `OP_REGISTRY` variable. Plugins may register new operations. A convenient way for developers is to use specific *decorators* that automatically register an annotated Python function or class and add additional meta-information to the operation registration's `OpMetaInfo` object. They are

- `@op(properties)` registers the function as operation and adds meta-information *properties* to the operation.
- `@op_input(name, properties)` adds extra meta-information *properties* to a named function input (argument)
- `@op_output(name, properties)` adds extra meta-information *properties* to a named function output
- `@op_return(name, properties)` adds extra meta-information *properties* to a single function output (return value)

Note that if a Python function defines an argument named `monitor`, it will not be considered as an operation input. Instead it is assumed that it is a monitor instance passed in by the CCI Toolbox, e.g. when invoking an operation from the command-line or if an operation is performed as part of a workflow as described in the next section.

7.5 Workflow Management

Many analyses on climate data can be decomposed into some sequential steps that perform some fundamental operation. To make such recurring chains of operations reusable and reproduceable, the CCI Toolbox contains a simple but powerful concept which is implemented in the `cate.core.workflow` module.

A *workflow* is a network or to be more specific, a directed acyclic graph of *steps*. A step execution may invoke a registered *operation* (see section *Operation Management*), may evaluate a simple Python expressions, may spawn an external process, and invoke another workflow.

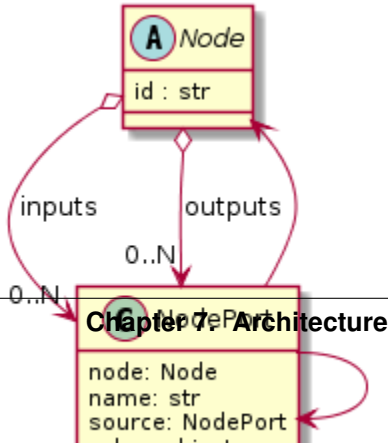
An great advantage of using workflows instead of, e.g. programming scripts, is that that the invocation of steps is controlled and monitored by the CCI Toolbox framework. This allows for task cancellation by users, task progress reporting, input/output validation. Workflows can be composed by a dedicated GUI or written by hand in a text editor, e.g. in JSON, YAML or XML format. Workflow steps can even be used to automatically ingest provenance information into the dataset outputs for processing traceability and later data history reconstruction.

Fig. 7.4 shows the types and relationships in the `cate.core.workflow` module:

- A `Node` has zero or more *inputs* and zero or more *outputs* and can be invoked.
- A `Workflow` is a `Node` that is composed of `Step` objects.
- A `Step` is a `Node` that is part of a `Workflow` and performs some kind of data processing.
- A `OpStep` is a `Step` that invokes an `Operation`.
- An `ExpressionStep` is a `Step` that executes a Python expression string.
- A `WorkflowStep` is a `Step` that executes a `Workflow` loaded from an external (JSON) resource.

Like the `Operation`, every `Node` has an associated `OpMetaInfo` object specifying the node's signature in terms of its inputs and outputs. The actual `Node` inputs and outputs are modelled by the `NodePort` class. As shown in Fig. 7.5, a given node port belongs to exactly one `Node` and represents either a named input or output of that node. A node port has a name, a property `source`, and a property `value`. If `source` is set, it must be another `NodePort` that provides the actual port's value. The value of the `value` property can be basically anything that has an external (JSON) representation.

Workflow input ports are usually unspecified, but `value` may be set. Workflow output ports and a step's input ports are usually connected with output ports of other contained steps or inputs of the workflow via the `source` attribute. A step's output ports are usually unconnected because their `value` attribute is set by a step's concrete implementation.



Similar to operations, users can run workflows from the command-line (see section *Command-Line Interface*), or may be invoked from the WebAPI (see

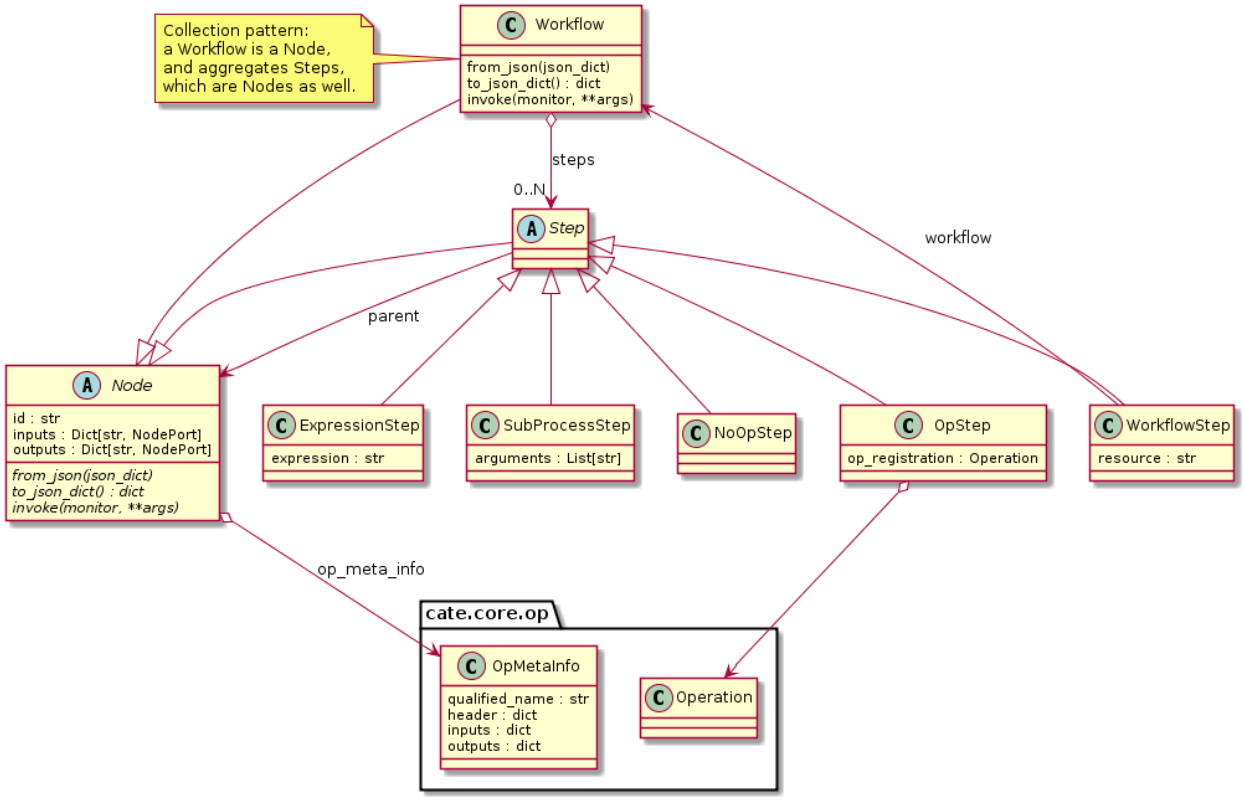


Fig. 7.4: Workflow, Node, Step

Fig. 7.1) due to a GUI request. The CCI Toolbox will always call workflows with a `Monitor` instance (see section *Task Monitoring*) and therefore sub-monitors will be passed to the contained steps.

The `workflow` module is independent of any other CCI Toolbox module so that it may later be replaced by a more advanced workflow management system.

7.6 Object Input/Output

The `objectio` module provides two generic functions for Python object input and output:

- `read_object(file, format)` reads an object from a file with optional format name, if known.
- `write_object(obj, file, format)` writes an object to a file with a given format.

The module defines the abstract base class `ObjectIO` which is implemented by classes that read Python objects from files and write them into files. `ObjectIO` instances represent a file format and the Python object types that they can read from and write to files of that format. Therefore they can make a guess how suitable they are for reading from a given file (method `read_fitness(file)`) or writing an object to a file (method `write_fitness(obj)`).

`ObjectIO` instances are registered in the `OBJECT_IO_REGISTRY` singleton which can be extended by plug-ins.

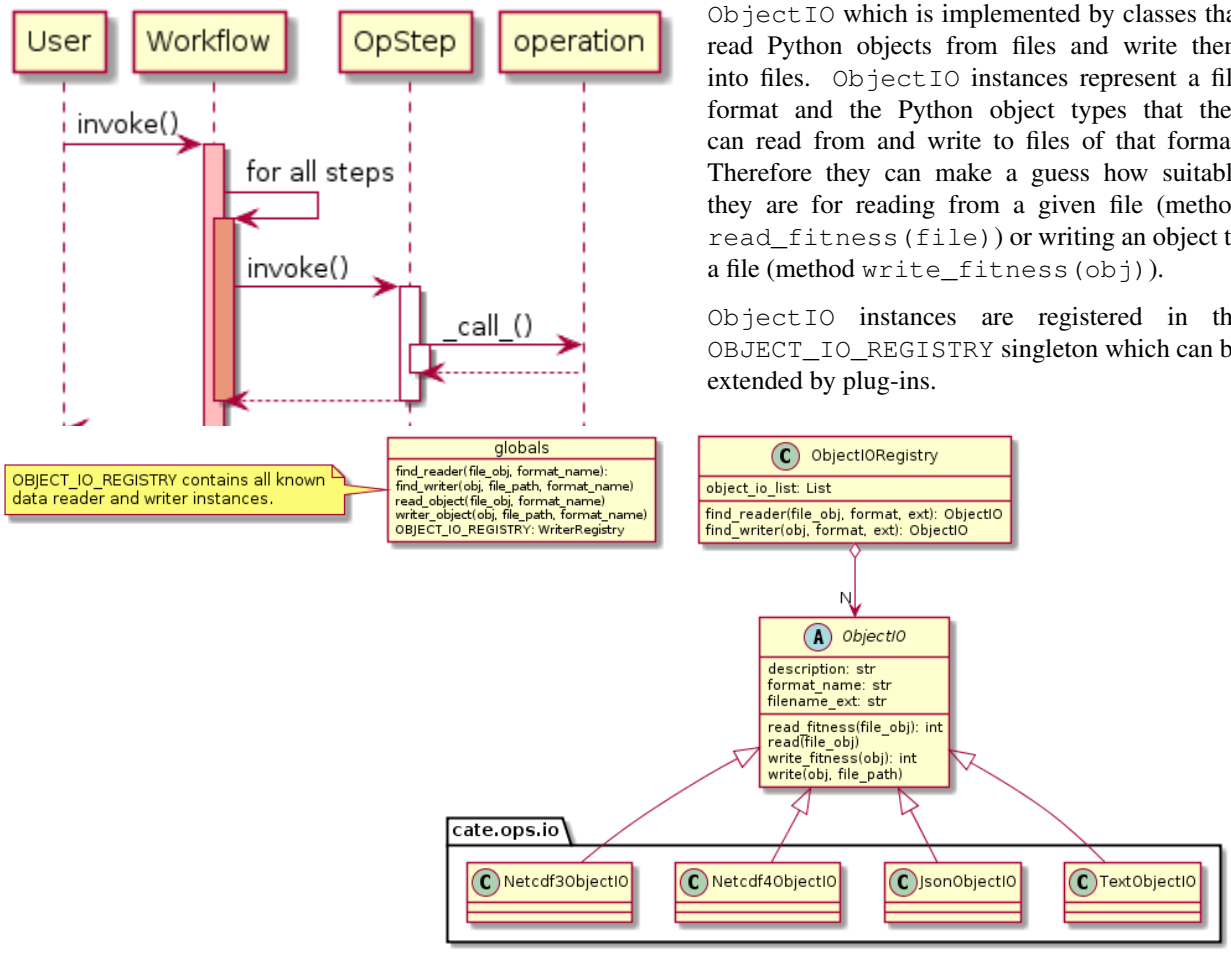


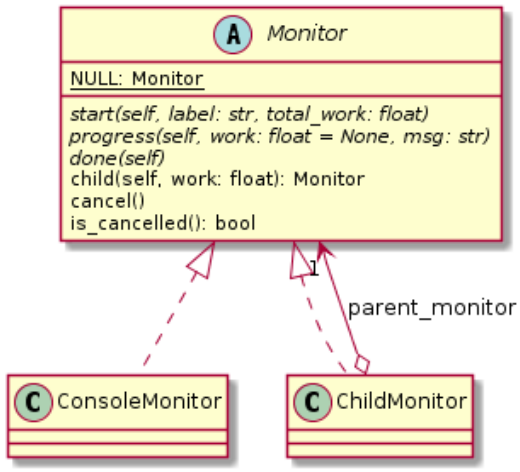
Fig. 7.7: ObjectIO and some of its implementations

7.7 Task Monitoring

The `monitor` module defines the abstract base class `Monitor` that that may be used by functions and methods that offer support for observation and control of long-running tasks. Concrete `Monitor``s may be implemented by API clients for a given context. The ``monitor` module defines two useful implementations.

- `ConsoleMonitor`: a monitor that is used by the command-line interface
- `ChildMonitor`: a sub-monitor that can be passed to sub-tasks called from the current task

In addition, the `Monitor.NONE` object, is a monitor singleton that basically does nothing. It is used instead of passing `None` into methods that don't require monitoring but expect a non-`None` argument value.



7.8 Command-Line Interface

The primary user interface of the CCI Toolbox' Python core is a command-line interface (CLI) executable named `cate`.

The CLI can be used to list available data sources and to synchronise subsets of remote data store contents on the user's computer to make them available to the CCI Toolbox. It also allows for listing available operations as well as running operations and workflows.

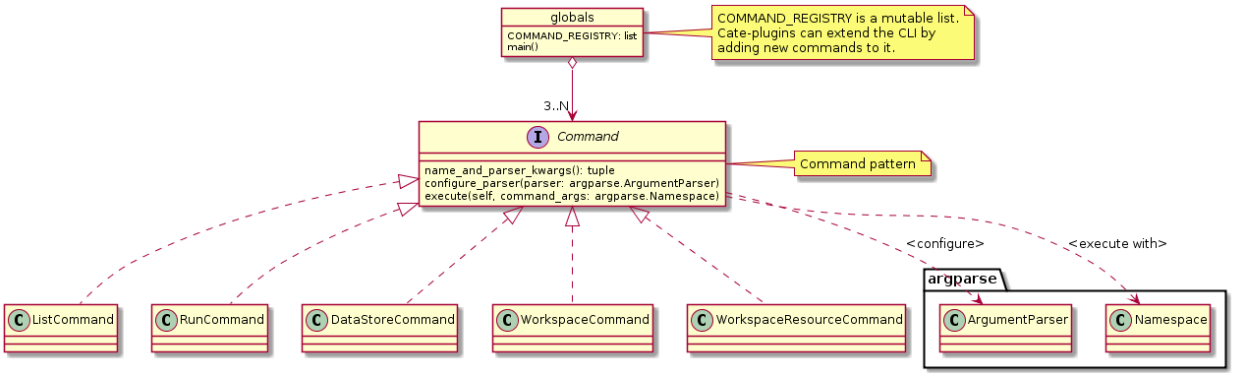


Fig. 7.9: CLI Command and sub-classes

The CLI uses (sub-)commands for specific functionality. The most important commands are

- `run` to run an operation or a *Workflow JSON* file with given arguments.
- `ds` to manage data sources and to synchronise remote data sources with locally cached versions of it.
- `op` to list and display details about available operations.

- `ws` to manage user *workspaces*.
- `res` to add, compute, modify, and display *resources* within the current user workspace.

Each command has its own set of options and arguments and can display help when used with the option `--help or `-h`.

Plugins can easily add new CLI commands to the CCI Toolbox by implementing a new `Command` class and registering it in the `COMMAND_REGISTRY` singleton.

7.9 Plugin Concept

A CCI Toolbox *plugin* is actually any Python module that extend one of the registry singletons introduced in the previous sections:

- Add a new `cate.core.ds.DataStore` object to `cate.core.ds.DATA_STORE_REGISTRY`
- Add a new `cate.core.op.Operation` object to `cate.core.op.OP_REGISTRY`
- Add a new `cate.core.objectio.ObjectIO` object to `cate.core.objectio.OBJECT_IO_REGISTRY`
- Add a new `cate.util.cli.Command` object to `cate.cli.COMMAND_REGISTRY`

It could also be a Python module that modifies or extends existing CCI Toolbox types by performing some controlled *monkey patching*.

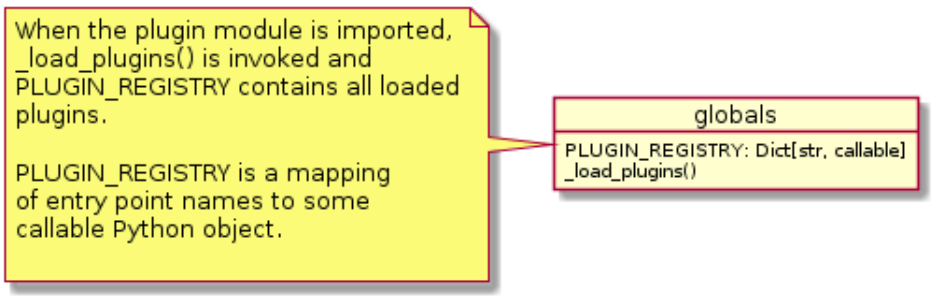


Fig. 7.10: The `plugin` module

The CCI Toolbox will call any plugin functions that are registered with the `cate_plugins` entry point of the standard Python `setuptools` module. These entry points can be easily provided in the plugin's `setup.py` file. The value of each entry point must be a no-arg initialisation function, which is called by the CCI Toolbox at given time. After successful initialisation the plugin is registered in the `PLUGIN_REGISTRY` singleton.

In fact the `cate.ds` and `cate.ops` packages of the CCI Toolbox Python core are such plugins registered with the same entry point:

```
setup(
    name="cate",
    version=__version__,
    description='ESA CCI Toolbox',
    license='MIT',
    author='ESA CCI Toolbox Development Team',
    packages=['cate'],
    entry_points={
        'console_scripts': [
```

(continues on next page)

(continued from previous page)

```
            'cate = cate.cli.main:main',
        ],
        'cate_plugins': [
            'cate_ops = cate.ops:cate_init',
            'cate_ds = cate.ds:cate_init',
        ],
    },
    ...
)
```

CHAPTER 8

API Reference

8.1 Datasets

cate.core.**find_data_sources**(*data_stores: Union[cate.core.ds.DataStore, typing.Sequence[cate.core.ds.DataStore]] = None, ds_id: str = None, query_expr: str = None*) → Sequence[cate.core.ds.DataSource]

Find data sources in the given data store(s) matching the given *id* or *query_expr*.

See also *open_dataset()*.

Return type Sequence

Parameters

- **data_stores** – If given these data stores will be queried. Otherwise all registered data stores will be used.
- **ds_id** (str) – A data source identifier.
- **query_expr** (str) – A query expression.

Returns All data sources matching the given constrains.

cate.core.**open_dataset**(*data_source: Union[cate.core.ds.DataSource, str], time_range: Union[typing.Tuple[str, str], typing.Tuple[datetime.datetime, datetime.datetime], typing.Tuple[datetime.date, datetime.date], str] = None, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, force_local: bool = False, local_ds_id: str = None, monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Any

Open a dataset from a data source.

Parameters

- **data_source** – A DataSource object or a string. Strings are interpreted as the identifier of an ECV dataset and must not be empty.

- **time_range** – An optional time constraint comprising start and end date. If given, it must be a `TimeRangeLike`.
- **region** – An optional region constraint. If given, it must be a `PolygonLike`.
- **var_names** – Optional names of variables to be included. If given, it must be a `VarNamesLike`.
- **force_local** (`bool`) – Optional flag for remote data sources only Whether to make a local copy of data source if it's not present
- **local_ds_id** (`str`) – Optional, fpr remote data sources only Local data source ID for newly created copy of remote data source
- **monitor** (`Monitor`) – A progress monitor

Returns An new dataset instance

8.2 Operations

8.2.1 Anomaly calculation

`cate.ops.`**`anomaly_internal`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Calculate anomaly using as reference data the mean of an optional region and time slice from the given dataset. If no time slice/spatial region is given, the operation will calculate anomaly using the mean of the whole dataset as the reference.

This is done for each data array in the dataset. :type monitor: `Monitor` :param ds: The dataset to calculate anomalies from :param time_range: Time range to use for reference data :param region: Spatial region to use for reference data :param monitor: a progress monitor. :return: The anomaly dataset

`cate.ops.`**`anomaly_external`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Calculate anomaly with external reference data, for example, a climatology. The given reference dataset is expected to consist of 12 time slices, one for each month.

The returned dataset will contain the variable names found in both - the reference and the given dataset. Names found in the given dataset, but not in the reference, will be dropped from the resulting dataset. The calculated anomaly will be against the corresponding month of the reference data. E.g. January against January, etc.

In case spatial extents differ between the reference and the given dataset, the anomaly will be calculated on the intersection.

Parameters

- **ds** – The dataset to calculate anomalies from
- **file** – Path to reference data file
- **transform** – Apply the given transformation before calculating the anomaly. For supported operations see help on 'ds_arithmetics' operation.
- **monitor** (`Monitor`) – a progress monitor.

Returns The anomaly dataset

8.2.2 Arithmetic

`cate.ops.`**`ds_arithmetics`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Do arithmetic operations on the given dataset by providing a list of arithmetic operations and the corresponding

constant. The operations will be applied to the dataset in the order in which they appear in the list. For example: 'log,+5,-2,/3,*2'

Currently supported arithmetic operations: log,log10,log2,log1p,exp,+,-,/,*

where: log - natural logarithm log10 - base 10 logarithm log2 - base 2 logarithm log1p - log(1+x) exp - the exponential

The operations will be applied element-wise to all arrays of the dataset.

Parameters

- **ds** – The dataset to which to apply arithmetic operations
- **op** – A comma separated list of arithmetic operations to apply
- **monitor** (Monitor) – a progress monitor.

Returns The dataset with given arithmetic operations applied

8.2.3 Averaging

cate.ops.**long_term_average**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Create a 'mean over years' dataset by averaging the values of the given input dataset over all years. The output is a climatological dataset with the same resolution as the input dataset. E.g. a daily input dataset will create a daily climatology consisting of 365 days, a monthly input dataset will create a monthly climatology, etc.

Seasonal input datasets must have matching seasons over all years denoted by the same date each year. E.g., first date of each quarter. The output dataset will then be a seasonal climatology where each season is denoted with the same date as in the input dataset.

For further information on climatological datasets, see http://cfconventions.org/cf-conventions/v1.6.0/cf-conventions.html#climatological-statistics

Parameters

- **ds** – A dataset to average
- **var** – If given, only these variables will be preserved in the resulting dataset
- **monitor** (Monitor) – A progress monitor

Returns A climatological long term average dataset

cate.ops.**temporal_aggregation**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Perform aggregation of dataset according to the given method and output resolution.

Note that the operation does not perform weighting. Depending on the combination of input and output resolutions, as well as aggregation method, the resulting dataset might yield unexpected results.

Resolution 'month' will result in a monthly dataset with each month denoted by its first date. Resolution 'season' will result in a dataset aggregated to DJF, MAM, JJA, SON seasons, each denoted by the first date of the season.

The operation also works with custom resolution strings, see: http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases If `custom_resolution` is provided, it will override `output_resolution`.

Some examples: 'QS-JUN' produces an output dataset on a quarterly resolution where the year ends in 1st of June and each quarter is denoted by its first date '8MS' produces an output dataset on an eight-month resolution where each period is denoted by the first date. Note that such periods will not be consistent over years. '8D' produces a dataset on an eight day resolution

Parameters

- **ds** – Dataset to aggregate
- **method** – Aggregation method
- **output_resolution** – Desired temporal resolution of the output dataset
- **custom_resolution** – Custom temporal resolution, overrides output_resolution

Returns Aggregated dataset

8.2.4 Coregistration

cate.ops.**coregister**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Perform coregistration of two datasets by resampling the slave dataset unto the grid of the master. If upsampling has to be performed, this is achieved using interpolation, if downsampling has to be performed, the pixels of the slave dataset are aggregated to form a coarser grid.

The returned dataset will contain the lat/lon intersection of provided master and slave datasets, resampled unto the master grid frequency.

This operation works on datasets whose spatial dimensions are defined on pixel-registered and equidistant in lat/lon coordinates grids. E.g., data points define the middle of a pixel and pixels have the same size across the dataset.

This operation will resample all variables in a dataset, as the lat/lon grid is defined per dataset. It works only if all variables in the dataset have lat and lon as dimensions.

For an overview of downsampling/upsampling methods used in this operation, please see https://github.com/CAB-LAB/gridtools

Whether upsampling or downsampling has to be performed is determined automatically based on the relationship of the grids of the provided datasets.

Parameters

- **ds_master** – The dataset whose grid is used for resampling
- **ds_slave** – The dataset that will be resampled
- **method_us** – Interpolation method to use for upsampling.
- **method_ds** – Interpolation method to use for downsampling.
- **monitor** (Monitor) – a progress monitor.

Returns The slave dataset resampled on the grid of the master

8.2.5 Correlation

cate.ops.**pearson_correlation_scalar**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Do product moment Pearson's correlation analysis.

Performs a simple correlation analysis on two timeseries and returns a correlation coefficient and the corresponding p_value.

Positive correlation implies that as x grows, so does y. Negative correlation implies that as x increases, y decreases.

For more information how to interpret the results, see here, and here.

Parameters

- **ds_x** – The 'x' dataset
- **ds_y** – The 'y' dataset
- **var_x** – Dataset variable to use for correlation analysis in the 'variable' dataset
- **var_y** – Dataset variable to use for correlation analysis in the 'dependent' dataset
- **monitor** (Monitor) – a progress monitor.

Returns {'corr_coef': correlation coefficient, 'p_value': probability value}

cate.ops.**pearson_correlation**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Do product moment Pearson's correlation analysis.

Perform Pearson correlation on two datasets and produce a lon/lat map of correlation coefficients and the correspoding p_values.

In case two 3D lon/lat/time datasets are provided, pixel by pixel correlation will be performed. It is also possible two pro Perform Pearson correlation analysis on two time/lat/lon datasets and produce a lat/lon map of correlation coefficients and p_values of underlying timeseries in the provided datasets.

The lat/lon definition of both datasets has to be the same. The length of the time dimension should be equal, but not neccessarily have the same definition. E.g., it is possible to correlate different times of the same area.

There are 'x' and 'y' datasets. Positive correlations imply that as x grows, so does y. Negative correlations imply that as x increases, y decreases.

For more information how to interpret the results, see here, and here.

Parameters

- **ds_x** – The 'x' dataset
- **ds_y** – The 'y' dataset
- **var_x** – Dataset variable to use for correlation analysis in the 'variable' dataset
- **var_y** – Dataset variable to use for correlation analysis in the 'dependent' dataset
- **monitor** (Monitor) – a progress monitor.

Returns a dataset containing a map of correlation coefficients and p_values

8.2.6 Data Frame

cate.ops.**data_frame_min**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Select the first record of a data frame for which the given variable value is minimal.

Parameters

- **df** – The data frame or dataset.
- **var** – The variable.

Returns A new, one-record data frame.

cate.ops.**data_frame_max**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Select the first record of a data frame for which the given variable value is maximal.

Parameters

- **df** – The data frame or dataset.

- **var** – The variable.

Returns A new, one-record data frame.

cate.ops.**data_frame_query**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Select records from the given data frame where the given conditional query expression evaluates to "True".

If the data frame *df* contains a geometry column (a `GeoDataFrame` object), then the query expression *query_expr* can also contain geometric relationship tests, for example the expression `"population > 100000 and @within('-10, 34, 20, 60')"` could be used on a data frame with the *population* and a *geometry* column to query for larger cities in West-Europe.

The geometric relationship tests are * `@almost_equals(geom)` - does a feature's geometry almost equal the given `geom`; * `@contains(geom)` - does a feature's geometry contain the given `geom`; * `@crosses(geom)` - does a feature's geometry cross the given `geom`; * `@disjoint(geom)` - does a feature's geometry not at all intersect the given `geom`; * `@intersects(geom)` - does a feature's geometry intersect with given `geom`; * `@touches(geom)` - does a feature's geometry have a point in common with given `geom` but does not intersect it; * `@within(geom)` - is a feature's geometry contained within given `geom`.

The *geom* argument may be a point `"<lon>, <lat>"` text string, a bounding box `"<lon1>, <lat1>, <lon2>, <lat2>"` text, or any valid geometry WKT.

Parameters

- **df** – The data frame or dataset.
- **query_expr** – The conditional query expression.

Returns A new data frame.

8.2.7 Input / Output

cate.ops.**open_dataset**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Open a dataset from a data source identified by *ds_name*.

Parameters

- **ds_name** – The name of data source. This parameter has been deprecated, please use *ds_id* instead.
- **ds_id** – The identifier for the data source.
- **time_range** – Optional time range of the requested dataset
- **region** – Optional spatial region of the requested dataset
- **var_names** – Optional names of variables of the requested dataset
- **normalize** – Whether to normalize the dataset's geo- and time-coding upon opening. See operation `normalize`.
- **force_local** – Whether to make a local copy of remote data source if it's not present
- **local_ds_id** – Optional local identifier for newly created local copy of remote data source. Used only if force_local=True.
- **monitor** (`Monitor`) – A progress monitor

Returns An new dataset instance.

cate.ops.**save_dataset**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Save a dataset to NetCDF file.

Parameters

- **ds** – The dataset
- **file** – File path
- **format** – NetCDF format flavour, one of 'NETCDF4', 'NETCDF4_CLASSIC', 'NETCDF3_64BIT', 'NETCDF3_CLASSIC'.
- **monitor** (Monitor) – a progress monitor.

cate.ops.**read_object**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Read a data object from a file.

Parameters

- **file** – The file path.
- **format** – Optional format name.

Returns The data object.

cate.ops.**write_object**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Write a data object to a file.

Parameters

- **obj** – The object to write.
- **file** – The file path.
- **format** – Optional format name.

Returns The data object.

cate.ops.**read_text**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Read a string object from a text file.

Parameters

- **file** – The text file path.
- **encoding** – Optional encoding, e.g. "utc-8".

Returns The string object.

cate.ops.**write_text**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Write an object as string to a text file.

Parameters

- **obj** – The data object.
- **file** – The text file path.
- **encoding** – Optional encoding, e.g. "utc-8".

cate.ops.**read_json**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Read a data object from a JSON text file.

Parameters

- **file** – The JSON file path.
- **encoding** – Optional encoding, e.g. "utc-8".

Returns The data object.

cate.ops.**write_json**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Write a data object to a JSON text file. Note that the data object must be JSON-serializable.

Parameters

- **obj** – A JSON-serializable data object.
- **file** – The JSON file path.
- **encoding** – Optional encoding, e.g. "utf-8".
- **indent** – indent used in the file, e.g. " " (two spaces).

cate.ops.**read_csv**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Read comma-separated values (CSV) from plain text file into a Pandas DataFrame.

Parameters

- **file** – The CSV file path.
- **delimiter** – Delimiter to use. If delimiter is None, will try to automatically determine this.
- **delim_whitespace** – Specifies whether or not whitespaces will be used as delimiter. If this option is set, nothing should be passed in for the delimiter parameter.
- **quotechar** – The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.
- **comment** – Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character.
- **index_col** – The name of the column that provides unique identifiers
- **more_args** – Other optional keyword arguments. Please refer to Pandas documentation of `pandas.read_csv()` function.

Returns The DataFrame object.

cate.ops.**read_geo_data_frame**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Reads geo-data from files with formats such as ESRI Shapefile, GeoJSON, GML.

Parameters

- **file** – Is either the absolute or relative path to the file to be opened.
- **crs** – Optional coordinate reference system. Must be given as CRS-WKT or EPSG string such as "EPSG:4326". The default value for GeoJSON standard is always "EPSG:4326".
- **more_args** – Other optional keyword arguments. Please refer to Python documentation of `fiona.open()` function.

Returns A `geopandas.GeoDataFrame` object

cate.ops.**read_netcdf**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Read a dataset from a netCDF 3/4 or HDF file.

Parameters

- **file** – The netCDF file path.
- **drop_variables** – List of variables to be dropped.
- **decode_cf** – Whether to decode CF attributes and coordinate variables.
- **normalize** – Whether to normalize the dataset's geo- and time-coding upon opening. See operation `normalize`.

- **decode_times** – Whether to decode time information (convert time coordinates to `datetime` objects).
- **engine** – Optional netCDF engine name.

`cate.ops.`**write_netcdf3**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Write a data object to a netCDF 3 file. Note that the data object must be netCDF-serializable.

Parameters

- **obj** – A netCDF-serializable data object.
- **file** – The netCDF file path.
- **engine** – Optional netCDF engine to be used

`cate.ops.`**write_netcdf4**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Write a data object to a netCDF 4 file. Note that the data object must be netCDF-serializable.

Parameters

- **obj** – A netCDF-serializable data object.
- **file** – The netCDF file path.
- **engine** – Optional netCDF engine to be used

8.2.8 Data visualization

`cate.ops.`**plot_map**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Create a geographic map plot for the variable given by dataset *ds* and variable name *var*.

Plots the given variable from the given dataset on a map with coastal lines. In case no variable name is given, the first encountered variable in the dataset is plotted. In case no *time* is given, the first time slice is taken. It is also possible to set extents of the plot. If no extents are given, a global plot is created.

The plot can either be shown using pyplot functionality, or saved, if a path is given. The following file formats for saving the plot are supported: eps, jpeg, jpg, pdf, pgf, png, ps, raw, rgba, svg, svgz, tif, tiff

Parameters

- **ds** – the dataset containing the variable to plot
- **var** – the variable's name
- **indexers** – Optional indexers into data array of *var*. The *indexers* is a dictionary or a comma-separated string of key-value pairs that maps the variable's dimension names to constant labels. e.g. "layer=4".
- **time** – time slice index to plot, can be a string "YYYY-MM-DD" or an integer number
- **region** – Region to plot
- **projection** – name of a global projection, see http://scitools.org.uk/cartopy/docs/v0.15/crs/projections.html
- **central_lon** – central longitude of the projection in degrees
- **title** – an optional title
- **contour_plot** – If true plot a filled contour plot of data, otherwise plots a pixelated colormesh

- **properties** – optional plot properties for Python matplotlib, e.g. "bins=512, range=(-1.5, +1.5)" For full reference refer to https://matplotlib.org/api/lines_api.html and https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.contourf.html
- **file** – path to a file in which to save the plot

Returns a matplotlib figure object or None if in IPython mode

cate.ops.**plot**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Create a 1D/line or 2D/image plot of a variable given by dataset *ds* and variable name *var*.

Parameters

- **ds** – Dataset or Dataframe that contains the variable named by *var*.
- **var** – The name of the variable to plot
- **indexers** – Optional indexers into data array of *var*. The *indexers* is a dictionary or a comma-separated string of key-value pairs that maps the variable's dimension names to constant labels. e.g. "lat=12.4, time='2012-05-02'".
- **title** – an optional plot title
- **properties** – optional plot properties for Python matplotlib, e.g. "bins=512, range=(-1.5, +1.5), label='Sea Surface Temperature'" For full reference refer to https://matplotlib.org/api/lines_api.html and https://matplotlib.org/devdocs/api/_as_gen/matplotlib.patches.Patch.html#matplotlib.patches.Patch
- **file** – path to a file in which to save the plot

Returns a matplotlib figure object or None if in IPython mode

cate.ops.**plot_data_frame**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Plot a data frame. This is a wrapper of pandas.DataFrame.plot() function. For further documentation please see http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html :type monitor: `Monitor` :param df: A pandas dataframe to plot :param plot_type: Plot type :param file: path to a file in which to save the plot :param kwargs: Keyword arguments to pass to the underlying

pandas.DataFrame.plot function

8.2.9 Resampling

cate.ops.**resample_2d**(*src*, *w*, *h*, *ds_method=54*, *us_method=11*, *fill_value=None*, *mode_rank=1*, *out=None*)
Resample a 2-D grid to a new resolution.

Parameters

- **src** – 2-D *ndarray*
- **w** – *int* New grid width
- **h** – *int* New grid height
- **ds_method** (`int`) – one of the *DS_* constants, optional Grid cell aggregation method for a possible downsampling
- **us_method** (`int`) – one of the *US_* constants, optional Grid cell interpolation method for a possible upsampling
- **fill_value** – *scalar*, optional If `None`, it is taken from **src** if it is a masked array, otherwise from *out* if it is a masked array, otherwise numpy's default value is used.

- **mode_rank** (int) – *scalar*, optional The rank of the frequency determined by the *ds_method* DS_MODE. One (the default) means most frequent value, zwo means second most frequent value, and so forth.
- **out** – 2-D *ndarray*, optional Alternate output array in which to place the result. The default is *None*; if provided, it must have the same shape as the expected output.

Returns An resampled version of the *src* array.

cate.ops.**downsample_2d**(*src*, *w*, *h*, *method=54*, *fill_value=None*, *mode_rank=1*, *out=None*)
Downsample a 2-D grid to a lower resolution by aggregating original grid cells.

Parameters

- **src** – 2-D *ndarray*
- **w** – *int* Grid width, which must be less than or equal to *src.shape[-1]*
- **h** – *int* Grid height, which must be less than or equal to *src.shape[-2]*
- **method** (int) – one of the *DS_* constants, optional Grid cell aggregation method
- **fill_value** – *scalar*, optional If None, it is taken from **src** if it is a masked array, otherwise from *out* if it is a masked array, otherwise numpy's default value is used.
- **mode_rank** (int) – *scalar*, optional The rank of the frequency determined by the *method* DS_MODE. One (the default) means most frequent value, zwo means second most frequent value, and so forth.
- **out** – 2-D *ndarray*, optional Alternate output array in which to place the result. The default is *None*; if provided, it must have the same shape as the expected output.

Returns A downsampled version of the *src* array.

cate.ops.**upsample_2d**(*src*, *w*, *h*, *method=11*, *fill_value=None*, *out=None*)
Upsample a 2-D grid to a higher resolution by interpolating original grid cells.

Parameters

- **src** – 2-D *ndarray*
- **w** – *int* Grid width, which must be greater than or equal to *src.shape[-1]*
- **h** – *int* Grid height, which must be greater than or equal to *src.shape[-2]*
- **method** (int) – one of the *US_* constants, optional Grid cell interpolation method
- **fill_value** – *scalar*, optional If None, it is taken from **src** if it is a masked array, otherwise from *out* if it is a masked array, otherwise numpy's default value is used.
- **out** – 2-D *ndarray*, optional Alternate output array in which to place the result. The default is *None*; if provided, it must have the same shape as the expected output.

Returns An upsampled version of the *src* array.

8.2.10 Subsetting

cate.ops.**select_var**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Filter the dataset, by leaving only the desired variables in it. The original dataset information, including original coordinates, is preserved.

Parameters

- **ds** – The dataset or dataframe from which to perform selection.

- **var** – One or more variable names to select and preserve in the dataset. All of these are valid 'var_name' 'var_name1,var_name2,var_name3' ['var_name1', 'var_name2']. One can also use wildcards when doing the selection. E.g., choosing 'var_name*' for selection will select all variables that start with 'var_name'. This can be used to select variables along with their auxiliary variables, to select all uncertainty variables, and so on.

Returns A filtered dataset

cate.ops.**subset_spatial**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Do a spatial subset of the dataset

Parameters

- **ds** – Dataset to subset
- **region** – Spatial region to subset
- **mask** – Should values falling in the bounding box of the polygon but not the polygon itself be masked with NaN.

Returns Subset dataset

cate.ops.**subset_temporal**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Do a temporal subset of the dataset.

Parameters

- **ds** – Dataset or dataframe to subset
- **time_range** – Time range to select

Returns Subset dataset

cate.ops.**subset_temporal_index**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Do a temporal indices based subset

Parameters

- **ds** – Dataset or dataframe to subset
- **time_ind_min** – Minimum time index to select
- **time_ind_max** – Maximum time index to select

Returns Subset dataset

8.2.11 Timeseries

cate.ops.**tseries_point**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)
Extract time-series from *ds* at given *lon*, *lat* position using interpolation *method* for each *var* given in a comma separated list of variables.

The operation returns a new timeseries dataset, that contains the point timeseries for all required variables with original variable meta-information preserved.

If a variable has more than three dimensions, the resulting timeseries variable will preserve all other dimensions except for lon/lat.

Parameters

- **ds** – The dataset from which to perform timeseries extraction.
- **point** – Point to extract, e.g. (lon,lat)

- **var** – Variable(s) for which to perform the timeseries selection if none is given, all variables in the dataset will be used.
- **method** – Interpolation method to use.

Returns A timeseries dataset

cate.ops.**tseries_mean**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Extract spatial mean timeseries of the provided variables, return the dataset that in addition to all the information in the given dataset contains also timeseries data for the provided variables, following naming convention 'var_name1_ts_mean'

If a data variable with more dimensions than time/lat/lon is provided, the data will be reduced by taking the mean of all data values at a single time position resulting in one dimensional timeseries data variable.

Parameters

- **ds** – The dataset from which to perform timeseries extraction.
- **var** – Variables for which to perform timeseries extraction
- **calculate_std** – Whether to calculate std in addition to mean
- **std_suffix** – Std suffix to use for resulting datasets, if std is calculated.
- **monitor** (Monitor) – a progress monitor.

Returns Dataset with timeseries variables

8.2.12 Misc

cate.ops.**normalize**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Normalize the geo- and time-coding upon opening the given dataset w.r.t. to a common (CF-compatible) convention used within Cate. This will maximize the compatibility of a dataset for usage with Cate's operations.

That is, * variables named "latitude" will be renamed to "lat"; * variables named "longitude" or "long" will be renamed to "lon";

Then, for equi-rectangular grids, * Remove 2D "lat" and "lon" variables; * Two new 1D coordinate variables "lat" and "lon" will be generated from original 2D forms.

Finally, it will be ensured that a "time" coordinate variable will be of type *datetime*.

Parameters **ds** – The dataset to normalize.

Returns The normalized dataset, or the original dataset, if it is already "normal".

cate.ops.**sel**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Return a new dataset with each array indexed by tick labels along the specified dimension(s).

This is a wrapper for the `xarray.sel()` function.

For documentation refer to xarray documentation at http://xarray.pydata.org/en/stable/generated/xarray.Dataset.sel.html#xarray.Dataset.sel

Parameters

- **ds** – The dataset from which to select.
- **point** – Optional geographic point given by longitude and latitude
- **time** – Optional time

- **indexers** – Keyword arguments with names matching dimensions and values given by scalars, slices or arrays of tick labels. For dimensions with multi-index, the indexer may also be a dict-like object with keys matching index level names.
- **method** – Method to use for inexact matches: * None: only exact matches * `pad` / `ffill`: propagate last valid index value forward * `backfill` / `bfill`: propagate next valid index value backward * `nearest` (default): use nearest valid index value

Returns A new Dataset with the same contents as this dataset, except each variable and dimension is indexed by the appropriate indexers. In general, each variable's data will be a view of the variable's data in this dataset.

`cate.ops.`**`from_dataframe`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Convert the given dataframe to an xarray dataset.

This is a wrapper for the `xarray.from_dataframe()` function.

For documentation refer to xarray documentation at http://xarray.pydata.org/en/stable/generated/xarray.Dataset.from_dataframe.html#xarray.Dataset.from_dataframe

Parameters **df** – Dataframe to convert

Returns A dataset created from the given dataframe

`cate.ops.`**`identity`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Return the given value. This operation can be useful to create constant resources to be used as input for other operations.

Parameters **value** – An arbitrary (Python) value.

`cate.ops.`**`literal`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Return the given value. This operation can be useful to create constant resources to be used as input for other operations.

Parameters **value** – An arbitrary (Python) literal.

`cate.ops.`**`pandas_fillna`**(**args*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*, ***kwargs*)

Return a new dataframe with NaN values filled according to the given value or method.

This is a wrapper for the `pandas.fillna()` function For additional keyword arguments and information refer to pandas documentation at http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html

Parameters

- **df** – The dataframe to fill
- **value** – Value to fill
- **method** – Method according to which to fill NaN. ffill/pad will propagate the last valid observation to the next valid observation. backfill/bfill will propagate the next valid observation back to the last valid observation.
- **limit** – Maximum number of NaN values to forward/backward fill.

Returns A dataframe with nan values filled with the given value or according to the given method.

8.3 Data Stores and Data Sources API

class `cate.core.`**`DataStore`**(*ds_id: str*, *title: str = None*, *is_local: bool = False*)

Represents a data store of data sources.

Parameters

- **ds_id** – Unique data store identifier.
- **title** – A human-readable tile.

id
Return the unique identifier for this data store.

invalidate()
Datastore might use a cached list of available dataset which can change in time. Resources managed by a datastore are external so we have to consider that they can be updated by other process. This method ask to invalidate the internal structure and synchronize it with the current status :return:

is_local
Whether this is a remote data source not requiring any internet connection when its `query()` method is called or the `open_dataset()` and `make_local()` methods on one of its data sources.

query(*ds_id: str = None*, *query_expr: str = None*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Sequence[cate.core.ds.DataSource]
Retrieve data sources in this data store using the given constraints.

Return type Sequence

Parameters

- **ds_id** (str) – Data source identifier.
- **query_expr** (str) – Query expression which may be used if *ìd* is unknown.
- **monitor** (Monitor) – A progress monitor.

Returns Sequence of data sources.

title
Return a human-readable tile for this data store.

class cate.core.**DataSource**
An abstract data source from which datasets can be retrieved.

cache_info
Return information about cached, locally available data sets. The returned dict, if any, is JSON-serializable.

data_store
The data store to which this data source belongs.

id
Data source identifier.

info_string
Return a textual representation of the meta-information about this data source. Useful for CLI / REPL applications.

make_local(*local_name: str, local_id: str = None, time_range: Union[typing.Tuple[str, str], typing.Tuple[datetime.datetime, datetime.datetime], typing.Tuple[datetime.date, datetime.date], str] = None, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Union[_ForwardRef('DataSource'), NoneType]
Turns this (likely remote) data source into a local data source given a name and a number of optional constraints.

If this is a remote data source, data will be downloaded and turned into a local data source which will be added to the data store named "local".

If this is already a local data source, a new local data source will be created by copying required data or data subsets.

The method returns the newly create local data source.

Parameters

- **local_name** (str) – A human readable name for the new local data source.
- **local_id** (str) – A unique ID to be used for the new local data source. If not given, a new ID will be generated.
- **time_range** – An optional time constraint comprising start and end date. If given, it must be a TimeRangeLike.
- **region** – An optional region constraint. If given, it must be a PolygonLike.
- **var_names** – Optional names of variables to be included. If given, it must be a VarNamesLike.
- **monitor** (Monitor) – A progress monitor.

Returns the new local data source

matches(*ds_id: str = None*, *query_expr: str = None*) → bool

Test if this data source matches the given *id* or *query_expr*. If neither *id* nor *query_expr* are given, the method returns True.

Return type bool

Parameters

- **ds_id** (str) – A data source identifier.
- **query_expr** (str) – A query expression. Currently, only simple search strings are supported.

Returns True, if this data sources matches the given *id* or *query_expr*.

meta_info

Return meta-information about this data source. The returned dict, if any, is JSON-serializable.

open_dataset(*time_range: Union[typing.Tuple[str, str], typing.Tuple[datetime.datetime, datetime.datetime], typing.Tuple[datetime.date, datetime.date], str] = None, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, protocol: str = None, monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Any

Open a dataset from this data source.

Parameters

- **time_range** – An optional time constraint comprising start and end date. If given, it must be a TimeRangeLike.
- **region** – An optional region constraint. If given, it must be a PolygonLike.
- **var_names** – Optional names of variables to be included. If given, it must be a VarNamesLike.
- **protocol** (str) – **Deprecated.** Protocol name, if None selected default protocol will be used to access data.

- **monitor** (Monitor) – A progress monitor.

Returns A dataset instance or None if no data is available for the given constraints.

schema
The data Schema for any dataset provided by this data source or None if unknown. Currently unused in cate.

status
Return information about data source accessibility

temporal_coverage(*monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Union[typing.Tuple[datetime.datetime, datetime.datetime], NoneType]
The temporal coverage as tuple (*start*, *end*) where *start* and *end* are UTC datetime instances.

Parameters **monitor** (Monitor) – a progress monitor.

Returns A tuple of (*start*, *end*) UTC datetime instances or None if the temporal coverage is unknown.

title
Human-readable data source title. The default implementation tries to retrieve the title from meta_info['title'].

variables_info
Return meta-information about the variables contained in this data source. The returned dict, if any, is JSON-serializable.

8.4 Operation Registration API

class cate.core.**Operation**(*wrapped_op: Callable*, *op_meta_info=None*)
An Operation comprises a wrapped callable (e.g. function, constructor, lambda form) and additional meta-information about the wrapped operation itself and its inputs and outputs.

Parameters

- **wrapped_op** – some callable object that will be wrapped.
- **op_meta_info** – operation meta information.

op_meta_info

Returns Meta-information about the operation, see cate.core.op.OpMetaInfo.

wrapped_op

Returns The actual operation object which may be any callable.

class cate.core.**OpMetaInfo**(*qualified_name: str*, *has_monitor: bool = False*, *header: dict = None*, *input_names: List[str] = None*, *inputs: Dict[str, Dict[str, Any]] = None*, *outputs: Dict[str, Dict[str, Any]] = None*)
Represents meta-information about an operation:

- *qualified_name*: a an ideally unique, qualified operation name
- *header*: dictionary of arbitrary operation attributes
- input: ordered dictionary of named inputs, each mapping to a dictionary of arbitrary input attributes
- output: ordered dictionary of named outputs, each mapping to a dictionary of arbitrary output attributes

Warning: *OpMetaInfo'* objects should be considered immutable. However, the dictionaries mentioned above are returned "as-is", mostly for performance reasons. Changing entries in these dictionaries directly may cause unwanted side-effects.

Parameters

- **qualified_name** – The operation's qualified name.
- **has_monitor** – Whether the operation supports a *Monitor* keyword argument named `monitor`.
- **header** – Header information dictionary.
- **input_names** – Input information dictionary.
- **inputs** – Input information dictionary.
- **outputs** – Output information dictionary.

MONITOR_INPUT_NAME = 'monitor'
The constant `'monitor'`, which is the name of an operation input that will receive a *Monitor* object as value.

RETURN_OUTPUT_NAME = 'return'
The constant `'return'`, which is the name of a single, unnamed operation output.

has_monitor

Returns `True` if the operation supports a *Monitor* value as additional keyword argument named `monitor`.

has_named_outputs

Returns `True` if the output value of the operation is expected be a dictionary-like mapping of output names to output values.

header

Returns Operation header attributes.

input_names
The input names in the order they have been declared.

Returns List of input names.

inputs
Mapping from an input name to a dictionary of properties describing the input.

Returns Named inputs.

outputs
Mapping from an output name to a dictionary of properties describing the output.

Returns Named outputs.

qualified_name

Returns Fully qualified name of the actual operation.

set_default_input_values(*input_values: Dict*)
If any missing input value in *input_values*, set value of "default_value" property, if it exists.

Parameters **input_values** (`Dict`) – The dictionary of input values that will be modified.

to_json_dict(*data_type_to_json=None*) → Dict[str, Any]
Return a JSON-serializable dictionary representation of this object. E.g. values of the *data_type'* property are converted from Python types to their string representation.

Returns A JSON-serializable dictionary

validate_input_values(*input_values: Dict*, *except_types=None*, *validation_exception_class=<class 'ValueError'>*)

Validate given *input_values* against the operation's input properties.

Parameters

- **input_values** (Dict) – The dictionary of input values.
- **except_types** – A set of types or None. If an input value's type is in this set, it will not be validated against the various input properties, such as data_type, nullable, value_set, value_range.
- **validation_exception_class** – The exception class to be used to raise exceptions if validation fails. Must derive from BaseException. Defaults to ValueError.

Raises **validation_error_class** – If *input_values* are invalid w.r.t. to the operation's input properties.

validate_output_values(*output_values: Dict*, *validation_exception_class: type = <class 'ValueError'>*)

Validate given *output_values* against the operation's output properties.

Parameters

- **output_values** (Dict) – The dictionary of output values.
- **validation_exception_class** (type) – The exception class to be used to raise exceptions if validation fails. Must derive from BaseException. Defaults to ValueError.

Raises **validation_error_class** – If *output_values* are invalid w.r.t. to the operation's output properties.

cate.core.**op**(*tags=UNDEFINED*, *version=UNDEFINED*, *res_pattern=UNDEFINED*, *deprecated=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

op is a decorator function that registers a Python function or class in the default operation registry or the one given by *registry*, if any. Any other keywords arguments in *header* are added to the operation's meta-information header. Classes annotated by this decorator must have callable instances.

When a function is registered, an introspection is performed. During this process, initial operation the meta-information header property *description* is derived from the function's docstring.

If any output of this operation will have its history information automatically updated, there should be version information found in the operation header. Thus it's always a good idea to add it to all operations:

```
@op(version='X.x')
```

Parameters

- **tags** – An optional list of string tags.
- **version** – An optional version string.
- **res_pattern** – An optional pattern that will be used to generate the names for data resources that are used to hold a reference to the objects returned by the operation and that are cached in a Cate workspace. Currently, the only pattern variable that is supported and that must be present is {index} which will be replaced by an integer number that is guaranteed to produce a unique resource name.

- **deprecated** – An optional boolean or a string. If a string is used, it should explain why the operation has been deprecated and which new operation to use instead. If set to `True`, the operation's doc-string should explain the deprecation.
- **registry** – The operation registry.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of operation.

cate.core.**op_input**(*input_name: str*, *default_value=UNDEFINED*, *units=UNDEFINED*, *data_type=UNDEFINED*, *nullable=UNDEFINED*, *value_set_source=UNDEFINED*, *value_set=UNDEFINED*, *value_range=UNDEFINED*, *script_lang=UNDEFINED*, *deprecated=UNDEFINED*, *position=UNDEFINED*, *context=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

`op_input` is a decorator function that provides meta-information for an operation input identified by *input_name*. If the decorated function or class is not registered as an operation yet, it is added to the default operation registry or the one given by *registry*, if any.

When a function is registered, an introspection is performed. During this process, initial operation meta-information input properties are derived for each positional and keyword argument named *input_name*:

Derived property	Source
position	The position of a positional argument, e.g. `2` for input `z` in `def f(x, y, z, c=2).`
default_value	The value of a keyword argument, e.g. `52.3` for input `latitude` from argument definition `latitude:float=52.3`
data_type	The type annotation type, e.g. `float` for input `latitude` from argument definition `latitude:float`

The derived properties listed above plus any of *value_set*, *value_range*, and any key-value pairs in *properties* are added to the input's meta-information. A key-value pair in *properties* will always overwrite the derived properties listed above.

Parameters

- **input_name** (`str`) – The name of an input.
- **default_value** – A default value.
- **units** – The geo-physical units of the input value.
- **data_type** – The data type of the input values. If not given, the type of any given, non-None *default_value* is used.
- **nullable** – If `True`, the value of the input may be `None`. If not given, it will be set to `True` if the *default_value* is `None`.
- **value_set_source** – The name of an input, which can be used to generate a dynamic value set.
- **value_set** – A sequence of the valid values. Note that all values in this sequence must be compatible with *data_type*.
- **value_range** – A sequence specifying the possible range of valid values.
- **script_lang** – The programming language for a parameter of data_type "str" that provides source code of a script, e.g. "python".

- **deprecated** – An optional boolean or a string. If a string is used, it should explain why the input has been deprecated and which new input to use instead. If set to `True`, the input's doc-string should explain the deprecation.
- **position** – The zero-based position of an input.
- **context** – If `True`, the value of the operation input will be a dictionary representing the current execution context. For example, when the operation is executed from a workflow, the dictionary will hold at least three entries: `workflow` provides the current workflow, `step` is the currently executed step, and `value_cache` which is a mapping from step identifiers to step outputs. If *context* is a string, the value of the operation input will be the result of evaluating the string as Python expression with the current execution context as local environment. This means, *context* may be an expression such as 'value_cache', 'workspace.base_dir', 'step', 'step.id'.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of the named output.
- **registry** – Optional operation registry.

cate.core.**op_output**(*output_name: str*, *data_type=UNDEFINED*, *deprecated=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

`op_output` is a decorator function that provides meta-information for an operation output identified by *output_name*. If the decorated function or class is not registered as an operation yet, it is added to the default operation registry or the one given by *registry*, if any.

If your function does not return multiple named outputs, use the *op_return()* decorator function. Note that:

```
@op_return(...)
def my_func(...):
    ...
```

if equivalent to:

```
@op_output('return', ...)
def my_func(...):
    ...
```

To automatically add information about cate, its version, this operation and its inputs, to this output, set 'add_history' to True:

```
@op_output('name', add_history=True)
```

Note that the operation should have version information added to it when add_history is True:

```
@op(version='X.x')
```

Parameters

- **output_name** (`str`) – The name of the output.
- **data_type** – The data type of the output value.
- **deprecated** – An optional boolean or a string. If a string is used, it should explain why the output has been deprecated and which new output to use instead. If set to `True`, the output's doc-string should explain the deprecation.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of the named output.
- **registry** – Optional operation registry.

cate.core.**op_return**(*data_type=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

op_return is a decorator function that provides meta-information for a single, anonymous operation return value (whose output name is "return"). If the decorated function or class is not registered as an operation yet, it is added to the default operation registry or the one given by *registry*, if any. Any other keywords arguments in *properties* are added to the output's meta-information.

When a function is registered, an introspection is performed. During this process, initial operation meta-information output properties are derived from the function's return type annotation, that is *data_type* will be e.g. float if a function is annotated as `def f(x, y) -> float: ...`.

The derived *data_type* property and any key-value pairs in *properties* are added to the output's meta-information. A key-value pair in *properties* will always overwrite a derived *data_type*.

If your function returns multiple named outputs, use the *op_output()* decorator function. Note that:

```
@op_return(...)
def my_func(...):
    ...
```

if equivalent to:

```
@op_output('return', ...)
def my_func(...):
    ...
```

To automatically add information about cate, its version, this operation and its inputs, to this output, set 'add_history' to True:

```
@op_return(add_history=True)
```

Note that the operation should have version information added to it when add_history is True:

```
@op(version='X.x')
```

Parameters

- **data_type** – The data type of the return value.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of the return value.
- **registry** – The operation registry.

8.5 Workflow API

class cate.core.**Workflow**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *node_id: str = None*)

A workflow of (connected) steps.

Parameters

- **op_meta_info** – Meta-information object of type *OpMetaInfo*.
- **node_id** – A node ID. If None, an ID will be generated.

find_node(*step_id: str*) → Union[_ForwardRef('Step'), NoneType]

Find a (child) node with the given *node_id*.

find_steps_to_compute(*step_id: str*) → List[_ForwardRef('Step')]
Compute the list of steps required to compute the output of the step with the given *step_id*. The order of the returned list is its execution order, with the step given by *step_id* is the last one.

Return type List

Parameters **step_id** (str) – The step to be computed last and whose output value is requested.

Returns a list of steps, which is never empty

invoke_steps(*steps: List[_ForwardRef('Step')], context: Dict = None, monitor_label: str = None, monitor=Monitor.NONE*) → None
Invoke just the given steps.

Parameters

- **steps** (List) – Selected steps of this workflow.
- **context** (Dict) – An optional execution context
- **monitor_label** (str) – An optional label for the progress monitor.
- **monitor** – The progress monitor.

classmethod load(*file_path_or_fp: Union[str, io.IOBase], registry=OP_REGISTRY*) → cate.core.workflow.Workflow
Load a workflow from a file or file pointer. The format is expected to be "Workflow JSON".

Parameters

- **file_path_or_fp** – file path or file pointer
- **registry** – Operation registry

Returns a workflow

remove_orphaned_sources(*removed_node: cate.core.workflow.Node*)
Remove all input/output ports, whose source is still referring to *removed_node*. :type removed_node: *Node* :param removed_node: A removed node.

classmethod sort_steps(*steps: List[_ForwardRef('Step')]*)
Sorts the list of workflow steps in the order they they can be executed.

sorted_steps
The workflow steps in the order they they can be executed.

steps
The workflow steps in the order they where added.

store(*file_path_or_fp: Union[str, io.IOBase]*) → None
Store a workflow to a file or file pointer. The format is "Workflow JSON".

Parameters **file_path_or_fp** – file path or file pointer

to_json_dict() → dict
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

update_sources() → None
Resolve unresolved source references in inputs and outputs.

update_sources_node_id(*changed_node: cate.core.workflow.Node*, *old_id: str*)
Update the source references of input and output ports from *old_id* to *new_id*.

class cate.core.**OpStep**(*operation*, *node_id: str = None*, *registry=OP_REGISTRY*)

An *OpStep* is a step node that invokes a registered operation of type *Operation*.

Parameters

- **operation** – A fully qualified operation name or operation object such as a class or callable.
- **registry** – An operation registry to be used to lookup the operation, if given by name.
- **node_id** – A node ID. If None, a unique ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)

Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)

Create a new step node instance from the given *json_dict*

class cate.core.**NoOpStep**(*inputs: dict = None*, *outputs: dict = None*, *node_id: str = None*)

A NoOpStep "performs" a no-op, which basically means, it does nothing. However, it might still be useful to define step that or duplicates or renames output values by connecting its own output ports with any of its own input ports. In other cases it might be useful to have a NoOpStep as a placeholder or blackbox for some other real operation that will be put into place at a later point in time.

Parameters

- **inputs** – input name to input properties mapping.
- **outputs** – output name to output properties mapping.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)

Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)

Create a new step node instance from the given *json_dict*

class cate.core.**ExpressionStep**(*expression: str*, *inputs=None*, *outputs=None*, *node_id=None*)

An ExpressionStep is a step node that computes its output from a simple (Python) *expression* string.

Parameters

- **expression** – A simple (Python) expression string.
- **inputs** – input name to input properties mapping.
- **outputs** – output name to output properties mapping.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)

Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)

Create a new step node instance from the given *json_dict*

class cate.core.**SubProcessStep**(*command: str*, *run_python: bool = False*, *env: Dict[str, str] = None*, *cwd: str = None*, *shell: bool = False*, *started_re: str = None*, *progress_re: str = None*, *done_re: str = None*, *inputs: Dict[str, Dict] = None*, *outputs: Dict[str, Dict] = None*, *node_id: str = None*)

A SubProcessStep is a step node that computes its output by a sub-process created from the given *program*.

Parameters

- **command** – A pattern that will be interpolated by input values to obtain the actual command (program with arguments) to be executed. May contain "{input_name}" fields which will be replaced by the actual input value converted to text. *input_name* must refer to a valid operation input name in *op_meta_info.input* or it must be the value of either the "write_to" or "read_from" property of another input's property map.
- **run_python** – If True, *command_line_pattern* refers to a Python script which will be executed with the Python interpreter that Cate uses.
- **cwd** – Current working directory to run the command line in.
- **env** – Environment variables passed to the shell that executes the command line.
- **shell** – Whether to use the shell as the program to execute.
- **started_re** – A regex that must match a text line from the process' stdout in order to signal the start of progress monitoring. The regex must provide the group names "label" or "total_work" or both, e.g. "(?P<label>w+)" or "(?P<total_work>d+)"
- **progress_re** – A regex that must match a text line from the process' stdout in order to signal process. The regex must provide group names "work" or "msg" or both, e.g. "(?P<msg>w+)" or "(?P<work>d+)"
- **done_re** – A regex that must match a text line from the process' stdout in order to signal the end of progress monitoring.
- **inputs** – input name to input properties mapping.
- **outputs** – output name to output properties mapping.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)
Create a new step node instance from the given *json_dict*

class cate.core.**WorkflowStep**(*workflow: cate.core.workflow.Workflow*, *resource: str*, *node_id: str = None*)
A *WorkflowStep* is a step node that invokes an externally stored `Workflow`.

Parameters

- **workflow** – The referenced workflow.
- **resource** – A resource (e.g. file path, URL) from which the workflow was loaded.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)
Create a new step node instance from the given *json_dict*

resource
The workflow's resource path (file path, URL).

workflow
The workflow.

class cate.core.**Step**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *node_id: str = None*)
A step is an inner node of a workflow.

Parameters **node_id** – A node ID. If None, a name will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*) → Union[_ForwardRef('Step'), NoneType]
Create a new step node instance from the given *json_dict*

parent_node
The node's ID.

persistent
Return whether this step is persistent. That is, if the current workspace is saved, the result(s) of a persistent step may be written to a "resource" file in the workspace directory using this step's ID as filename. The file format and filename extension will be chosen according to each result's data type. On next attempt to execute the step again, e.g. if a workspace is opened, persistent steps may read the "resource" file to produce the result rather than performing an expensive re-computation. :return: True, if so, False otherwise

to_json_dict()
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

class cate.core.**Node**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *node_id: str = None*)
Base class for all nodes including parent nodes (e.g. *Workflow*) and child nodes (e.g. *Step*).

All nodes have inputs and outputs, and can be invoked to perform some operation.

Inputs and outputs are exposed as attributes of the `input` and `output` properties and are both of type *NodePort*.

Parameters **node_id** – A node ID. If None, a name will be generated.

call(*context: Dict = None*, *monitor=Monitor.NONE*, *input_values: Dict = None*)
Calls this workflow with given *input_values* and returns the result.

The method does the following: 1. Set default_value where input values are missing in *input_values* 2. Validate the input_values using this workflows's meta-info 3. Set this workflow's input port values 4. Invoke this workflow with given *context* and *monitor* 5. Get this workflow's output port values. Named outputs will be returned as dictionary.

Parameters

- **context** (Dict) – An optional execution context. It will be used to automatically set the value of any node input which has a "context" property set to either `True` or a context expression string.
- **monitor** – An optional progress monitor.
- **input_values** (Dict) – The input values.

Returns The output values.

collect_predecessors(*predecessors: List[_ForwardRef('Node')]*, *excludes: List[_ForwardRef('Node')] = None*)
Collect this node (self) and preceding nodes in *predecessors*.

find_node(*node_id*) → Union[_ForwardRef('Node'), NoneType]
Find a (child) node with the given *node_id*.

find_port(*name*) → Union[_ForwardRef('NodePort'), NoneType]
Find port with given name. Output ports are searched first, then input ports. :param name: The port name :return: The port, or `None` if it couldn't be found.

id
The node's identifier.

inputs
The node's inputs.

invoke(*context: Dict = None*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*) → None
Invoke this node's underlying operation with input values from `input`. Output values in `output` will be set from the underlying operation's return value(s).

Parameters

- **context** (`Dict`) – An optional execution context.
- **monitor** (`Monitor`) – An optional progress monitor.

max_distance_to(*other_node: cate.core.workflow.Node*) → int
If *other_node* is a source of this node, then return the number of connections from this node to *node*. If it is a direct source return `1`, if it is a source of the source of this node return `2`, etc. If *other_node* is this node, return 0. If *other_node* is not a source of this node, return -1.

Return type `int`

Parameters **other_node** – The other node.

Returns The distance to *other_node*

op_meta_info
The node's operation meta-information.

outputs
The node's outputs.

parent_node
The node's parent node or `None` if this node has no parent.

requires(*other_node: cate.core.workflow.Node*) → bool
Does this node require *other_node* for its computation? Is *other_node* a source of this node?

Return type `bool`

Parameters **other_node** – The other node.

Returns `True` if this node is a target of *other_node*

root_node
The root_node node.

set_id(*node_id: str*) → None
Set the node's identifier.

Parameters **node_id** (`str`) – The new node identifier. Must be unique within a workflow.

to_json_dict()
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

update_sources()
Resolve unresolved source references in inputs and outputs.

update_sources_node_id(*changed_node: cate.core.workflow.Node*, *old_id: str*)
Update the source references of input and output ports from *old_id* to *new_id*.

class cate.core.**NodePort**(*node: cate.core.workflow.Node*, *name: str*)
Represents a named input or output port of a *Node*.

to_json (*force_dict=False*)
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

update_source ()
Resolve this node port's source reference, if any.

If the source reference has the form *node-id.port-name* then *node-id* must be the ID of the workflow or any contained step and *port-name* must be a name either of one of its input or output ports.

If the source reference has the form *.port-name* then *node-id* will refer to either the current step or any of its parent nodes that contains an input or output named *port-name*.

If the source reference has the form *node-id* then *node-id* must be the ID of the workflow or any contained step which has exactly one output.

If *node-id* refers to a workflow, then *port-name* is resolved first against the workflow's inputs followed by its outputs. If *node-id* refers to a workflow's step, then *port-name* is resolved first against the step's outputs followed by its inputs.

Raises **ValueError** – if the source reference is invalid.

update_source_node_id (*node: cate.core.workflow.Node*, *old_node_id: str*) → None
A node identifier has changed so we update the source references and clear the source of input and output ports from *old_node_id* to *node.id*.

Parameters

- **node** (*Node*) – The node whose identifier changed.
- **old_node_id** (`str`) – The former node identifier.

8.6 Task Monitoring API

class cate.core.**Monitor**
A monitor is used to both observe and control a running task.

The `Monitor` class is an abstract base class for concrete monitors. Derived classes must implement the following three abstract methods: *start()*, *progress()*, and *done()*. Derived classes must implement also the following two abstract methods, if they want cancellation support: *cancel()* and *is_cancelled()*.

Pass `Monitor.NONE` to functions that expect a monitor instead of passing `None`.

Given here is an example of how progress monitors should be used by functions::

```
def long_running_task(a, b, c, monitor):
    with monitor.starting('doing a long running task', total_work=100)
        # do 30% of the work here
        monitor.progress(work=30)
        # do 70% of the work here
        monitor.progress(work=70)
```

If a function makes calls to other functions that also support a monitor, a *child-monitor* is used::

```
def long_running_task(a, b, c, monitor):
    with monitor.starting('doing a long running task', total_work=100)
        # let other_task do 30% of the work
        other_task(a, b, c, monitor=monitor.child(work=30))
```

(continues on next page)

(continued from previous page)

```
        # let other_task do 70% of the work
        other_task(a, b, c, monitor=monitor.child(work=70))
```

NONE = Monitor.NONE
A valid monitor that effectively does nothing. Use `Monitor.NONE` it instead of passing `None` to functions and methods that expect an argument of type `Monitor`.

cancel()
Request the task to be cancelled. This method will be usually called from the code that created the monitor, not by users of the monitor. For example, a GUI could create the monitor due to an invocation of a long-running task, and then the user wishes to cancel that task. The default implementation does nothing. Override to implement something useful.

check_for_cancellation()
Checks if the monitor has been cancelled and raises a `Cancellation` in that case.

child(*work: float = 1*) → cate.util.monitor.Monitor
Return a child monitor for the given partial amount of *work*.

Parameters **work** (`float`) – The partial amount of work.

Returns a sub-monitor

done()
Call to signal that a task has been done.

is_cancelled() → bool
Check if there is an external request to cancel the current task observed by this monitor.

Users of a monitor shall frequently call this method and check its return value. If cancellation is requested, they should politely exit the current processing in a proper way, e.g. by cleaning up allocated resources. The default implementation returns `False`. Subclasses shall override this method to return `True` if a task cancellation request was detected.

Returns `True` if task cancellation was requested externally. The default implementation returns `False`.

observing(*label: str*)
A context manager for easier use of progress monitors. Observes a `dask` task and reports back to the monitor.

Parameters **label** (`str`) – Passed to the monitor's `start` method

Returns

progress(*work: float = None*, *msg: str = None*)
Call to signal that a task has mode some progress.

Parameters

- **work** (`float`) – The incremental amount of work.
- **msg** (`str`) – A detail message.

start(*label: str*, *total_work: float = None*)
Call to signal that a task has started.

Note that *label* and *total_work* are not passed to `__init__`, because they are usually not known at constructions time. It is the responsibility of the task to derive the appropriate values for these.

Parameters

- **label** (`str`) – A task label

- **total_work** (float) – The total amount of work

starting(*label: str*, *total_work: float = None*)
A context manager for easier use of progress monitors. Calls the monitor's start method with *label* and *total_work*. Will then take care of calling *Monitor.done()*.

Parameters

- **label** (str) – Passed to the monitor's start method
- **total_work** (float) – Passed to the monitor's start method

Returns

class cate.core.**ConsoleMonitor**(*stay_in_line=False*, *progress_bar_size=1*)
A simple console monitor that directly writes to sys.stdout and detects user cancellation requests via CTRL+C.

Parameters

- **stay_in_line** – If True, the text written out will stay in the same line.
- **progress_bar_size** – If > 1, a progress monitor of max. *progress_bar_size* characters will be written to the console.

cancel()
Request the task to be cancelled. This method will be usually called from the code that created the monitor, not by users of the monitor. For example, a GUI could create the monitor due to an invocation of a long-running task, and then the user wishes to cancel that task. The default implementation does nothing. Override to implement something useful.

done()
Call to signal that a task has been done.

is_cancelled() → bool
Check if there is an external request to cancel the current task observed by this monitor.

Users of a monitor shall frequently call this method and check its return value. If cancellation is requested, they should politely exit the current processing in a proper way, e.g. by cleaning up allocated resources. The default implementation returns False. Subclasses shall override this method to return True if a task cancellation request was detected.

Returns True if task cancellation was requested externally. The default implementation returns False.

progress(*work: float = None*, *msg: str = None*)
Call to signal that a task has mode some progress.

Parameters

- **work** (float) – The incremental amount of work.
- **msg** (str) – A detail message.

start(*label: str*, *total_work: float = None*)
Call to signal that a task has started.

Note that *label* and *total_work* are not passed to __init__, because they are usually not known at constructions time. It is the responsibility of the task to derive the appropriate values for these.

Parameters

- **label** (str) – A task label
- **total_work** (float) – The total amount of work

CHAPTER 9

Detailed Design

This chapter provides the CCI Toolbox detailed design documentation. Its is generated from the docstrings that are extensively used throughout the Python code.

The documentation is generated for individual modules. Note that this modularisation reflects the effective, internal (and physical) structure of the Python code. This is not the official *API*, which comprises a relatively stable subset of the components, types, interfaces, and variables describes here and is described in chapter *API Reference*.

Each top level module documentation in the following sections provides a sub-section *Description* that provides the module's purpose, contents, and possibly its usage. Module descriptions may link into *Operation Specifications* for further explanation and traceability of the detailed design. An optional sub-section *Technical Requirements* provides a mapping from URD requirements to technical requirements and software features that drove the design of a module. If available, links to verifying unit-tests are given in sub-sections called *Verification*. The sub-section *Components* lists all documented, non-private components of a module, including variables, functions, and classes.

9.1 Module `cate.core.ds`

9.1.1 Description

This module provides Cate's data access API.

9.1.2 Technical Requirements

Query data store

Description Allow querying registered ECV data stores using a simple function that takes a set of query parameters and returns data source identifiers that can be used to open respective ECV dataset in the Cate.

URD-Source

- CCIT-UR-DM0006: Data access to ESA CCI

- CCIT-UR-DM0010: The data module shall have the means to attain meta-level status information per ECV type
- CCIT-UR-DM0013: The CCI Toolbox shall allow filtering

Add data store

Description Allow adding of user defined data stores specifying the access protocol and the layout of the data. These data stores can be used to access datasets.

URD-Source

- CCIT-UR-DM0011: Data access to non-CCI data

Open dataset

Description Allow opening an ECV dataset given an identifier returned by the *data store query*. The dataset returned complies to the Cate common data model. The dataset to be returned can optionally be constrained in time and space.

URD-Source

- CCIT-UR-DM0001: Data access and input
- CCIT-UR-DM0004: Open multiple inputs
- CCIT-UR-DM0005: Data access using different protocols>
- CCIT-UR-DM0007: Open single ECV
- CCIT-UR-DM0008: Open multiple ECV
- CCIT-UR-DM0009: Open any ECV
- CCIT-UR-DM0012: Open different formats

9.1.3 Verification

The module's unit-tests are located in test/test_ds.py and may be executed using `$ py.test test/test_ds.py --cov=cate/core/ds.py` for extra code coverage information.

9.1.4 Components

`cate.core.ds.DATA_STORE_REGISTRY = {'esa_cci_odp': EsaCciOdpDataStore (esa_cci_odp), 'loca`
The data data store registry of type *DataStoreRegistry*. Use it add new data stores to Cate.

exception `cate.core.ds.`**DataAccessError**(*message: str*, *source: Union[cate.core.ds.DataSource, cate.core.ds.DataStore, NoneType] = None*)
Exceptions produced by Cate's data stores and data sources instances, used to report any problems handling data.

exception `cate.core.ds.`**DataAccessWarning**
Warnings produced by Cate's data stores and data sources instances, used to report any problems handling data.

class `cate.core.ds.`**DataSource**
An abstract data source from which datasets can be retrieved.

cache_info
Return information about cached, locally available data sets. The returned dict, if any, is JSON-serializable.

data_store
The data store to which this data source belongs.

id
Data source identifier.

info_string
Return a textual representation of the meta-information about this data source. Useful for CLI / REPL applications.

make_local(*local_name: str, local_id: str = None, time_range: Union[typing.Tuple[str, str], typing.Tuple[datetime.datetime, datetime.datetime], typing.Tuple[datetime.date, datetime.date], str] = None, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Union[_ForwardRef('DataSource'), NoneType]
Turns this (likely remote) data source into a local data source given a name and a number of optional constraints.

If this is a remote data source, data will be downloaded and turned into a local data source which will be added to the data store named "local".

If this is already a local data source, a new local data source will be created by copying required data or data subsets.

The method returns the newly create local data source.

Parameters

- **local_name** (str) – A human readable name for the new local data source.
- **local_id** (str) – A unique ID to be used for the new local data source. If not given, a new ID will be generated.
- **time_range** – An optional time constraint comprising start and end date. If given, it must be a TimeRangeLike.
- **region** – An optional region constraint. If given, it must be a PolygonLike.
- **var_names** – Optional names of variables to be included. If given, it must be a VarNamesLike.
- **monitor** (Monitor) – A progress monitor.

Returns the new local data source

matches(*ds_id: str = None*, *query_expr: str = None*) → bool
Test if this data source matches the given *id* or *query_expr*. If neither *id* nor *query_expr* are given, the method returns True.

Return type bool

Parameters

- **ds_id** (str) – A data source identifier.
- **query_expr** (str) – A query expression. Currently, only simple search strings are supported.

Returns True, if this data sources matches the given *id* or *query_expr*.

meta_info
Return meta-information about this data source. The returned dict, if any, is JSON-serializable.

open_dataset(*time_range: Union[typing.Tuple[str, str], typing.Tuple[datetime.datetime, datetime.datetime], typing.Tuple[datetime.date, datetime.date], str] = None, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, protocol: str = None, monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Any
Open a dataset from this data source.

Parameters

- **time_range** – An optional time constraint comprising start and end date. If given, it must be a `TimeRangeLike`.
- **region** – An optional region constraint. If given, it must be a `PolygonLike`.
- **var_names** – Optional names of variables to be included. If given, it must be a `VarNamesLike`.
- **protocol** (`str`) – **Deprecated.** Protocol name, if None selected default protocol will be used to access data.
- **monitor** (`Monitor`) – A progress monitor.

Returns A dataset instance or `None` if no data is available for the given constraints.

schema
The data `Schema` for any dataset provided by this data source or `None` if unknown. Currently unused in cate.

status
Return information about data source accessibility

temporal_coverage(*monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Union[typing.Tuple[datetime.datetime, datetime.datetime], NoneType]
The temporal coverage as tuple (*start*, *end*) where *start* and *end* are UTC `datetime` instances.

Parameters **monitor** (`Monitor`) – a progress monitor.

Returns A tuple of (*start*, *end*) UTC `datetime` instances or `None` if the temporal coverage is unknown.

title
Human-readable data source title. The default implementation tries to retrieve the title from `meta_info['title']`.

variables_info
Return meta-information about the variables contained in this data source. The returned dict, if any, is JSON-serializable.

class cate.core.ds.**DataSourceStatus**

Enum stating current state of Data Source accessibility.

- READY - data is complete and ready to use
- ERROR - data initialization process has been interrupted, causing that data source is incomplete or/and corrupted
- PROCESSING - data source initialization process is in progress.
- CANCELLED - data initialization process has been intentionally interrupted by user

class cate.core.ds.**DataStore**(*ds_id: str*, *title: str = None*, *is_local: bool = False*)

Represents a data store of data sources.

Parameters

- **ds_id** – Unique data store identifier.
- **title** – A human-readable tile.

id

Return the unique identifier for this data store.

invalidate()

Datastore might use a cached list of available dataset which can change in time. Resources managed by a datastore are external so we have to consider that they can be updated by other process. This method ask to invalidate the internal structure and synchronize it with the current status :return:

is_local

Whether this is a remote data source not requiring any internet connection when its `query()` method is called or the `open_dataset()` and `make_local()` methods on one of its data sources.

query(*ds_id: str = None*, *query_expr: str = None*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Sequence[cate.core.ds.DataSource]

Retrieve data sources in this data store using the given constraints.

Return type Sequence

Parameters

- **ds_id** (str) – Data source identifier.
- **query_expr** (str) – Query expression which may be used if *ìd* is unknown.
- **monitor** (Monitor) – A progress monitor.

Returns Sequence of data sources.

title

Return a human-readable tile for this data store.

class cate.core.ds.**DataStoreRegistry**

Registry of *DataStore* objects.

cate.core.ds.**find_data_sources**(*data_stores: Union[cate.core.ds.DataStore, typing.Sequence[cate.core.ds.DataStore]] = None*, *ds_id: str = None*, *query_expr: str = None*) → Sequence[cate.core.ds.DataSource]

Find data sources in the given data store(s) matching the given *id* or *query_expr*.

See also *open_dataset()*.

Return type Sequence

Parameters

- **data_stores** – If given these data stores will be queried. Otherwise all registered data stores will be used.
- **ds_id** (str) – A data source identifier.
- **query_expr** (str) – A query expression.

Returns All data sources matching the given constrains.

cate.core.ds.**format_cached_datasets_coverage_string**(*cache_coverage: dict*) → str
Return a textual representation of information about cached, locally available data sets. Useful for CLI / REPL applications. :rtype: str :type cache_coverage: dict :param cache_coverage: :return:

cate.core.ds.**format_variables_info_string**(*variables: dict*)
Return some textual information about the variables contained in this data source. Useful for CLI / REPL applications. :type variables: dict :param variables: :return:

cate.core.ds.**get_ext_chunk_sizes**(*ds: <Mock name='mock.Dataset' id='140450924242368'>, dim_names: Set[str] = None, init_value=0, map_fn=<built-in function max>, reduce_fn=None*) → Dict[str, int]
Get the external chunk sizes for each dimension of a dataset as provided in a variable's encoding object.

Return type Dict

Parameters

- **ds** – The dataset.
- **dim_names** (Set) – The names of dimensions of data variables whose external chunking should be collected.
- **init_value** (int) – The initial value (not necessarily a chunk size) for mapping multiple different chunk sizes.
- **map_fn** – The mapper function that maps a chunk size from a previous (initial) value.
- **reduce_fn** – The reducer function the reduces multiple mapped chunk sizes to a single one.

Returns A mapping from dimension name to external chunk sizes.

cate.core.ds.**get_spatial_ext_chunk_sizes**(*ds_or_path: Union[<Mock name='mock.Dataset' id='140450924242368'>, str]*) → Dict[str, int]
Get the spatial, external chunk sizes for the latitude and longitude dimensions of a dataset as provided in a variable's encoding object.

Return type Dict

Parameters **ds_or_path** – An xarray dataset or a path to file that can be opened by xarray.

Returns A mapping from dimension name to external chunk sizes.

cate.core.ds.**open_dataset**(*data_source: Union[cate.core.ds.DataSource, str], time_range: Union[typing.Tuple[str, str], typing.Tuple[datetime.datetime, datetime.datetime], typing.Tuple[datetime.date, datetime.date], str] = None, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, force_local: bool = False, local_ds_id: str = None, monitor: cate.util.monitor.Monitor = Monitor.NONE*) → Any
Open a dataset from a data source.

Parameters

- **data_source** – A DataSource object or a string. Strings are interpreted as the identifier of an ECV dataset and must not be empty.
- **time_range** – An optional time constraint comprising start and end date. If given, it must be a TimeRangeLike.
- **region** – An optional region constraint. If given, it must be a PolygonLike.

- **var_names** – Optional names of variables to be included. If given, it must be a VarNamesLike.
- **force_local** (bool) – Optional flag for remote data sources only Whether to make a local copy of data source if it's not present
- **local_ds_id** (str) – Optional, fpr remote data sources only Local data source ID for newly created copy of remote data source
- **monitor** (Monitor) – A progress monitor

Returns An new dataset instance

cate.core.ds.**open_xarray_dataset**(*paths, region: Union[<Mock name='mock.geometry.Polygon' id='140450924097152'>, typing.List[typing.Tuple[float, float]], str, typing.Tuple[float, float, float, float]] = None, var_names: Union[typing.List[str], str] = None, monitor: cate.util.monitor.Monitor = Monitor.NONE, **kwargs*) → <Mock name='mock.Dataset' id='140450924242368'>

Open multiple files as a single dataset. This uses dask. If each individual file of the dataset is small, one Dask chunk will coincide with one temporal slice, e.g. the whole array in the file. Otherwise smaller dask chunks will be used to split the dataset.

Parameters

- **paths** – Either a string glob in the form "path/to/my/files/*.nc" or an explicit list of files to open.
- **region** – Optional region constraint.
- **var_names** – Optional variable names constraint.
- **monitor** (Monitor) – Optional progress monitor.
- **kwargs** – Keyword arguments directly passed to xarray.open_mfdataset()

9.2 Module cate.core.op

9.2.1 Description

This modules provides classes and functions allowing to maintain *operations*. Operations can be called from the Cate command-line interface, may be referenced from within processing workflows, or may be called remotely e.g. from graphical user interface or web frontend. An operation (*Operation*) comprises a Python callable and some additional meta-information (OpMetaInfo) that allows for automatic input validation, input value conversion, monitoring, and inter-connection of multiple operations using processing workflows and steps.

Operations are registered in operation registries (*OpRegistry*), the default operation registry is accessible via the global, read-only OP_REGISTRY variable.

9.2.2 Technical Requirements

Operation registration, lookup, and invocation

Description Maintain a central place in the software that manages the available operations such as data processors, data converters, analysis functions, etc. Operations can be added, removed and retrieved. Operations are designed to be executed by the framework in a controlled way, i.e. an operation's

task can be monitored and cancelled, it's input and out values can be validated w.r.t. the operation's meta-information.

URD-Sources

- CCIT-UR-CR0001: Extensibility.
- CCIT-UR-E0002: dynamic extension of all modules at runtime, c) The Logic Module to introduce new processors
- CCIT-UR-LM0001: processor management allowing easy selection of tools and functionalities

---

Exploit Python language features

Description Exploit Python language to let API users express an operation in an intuitive form. For the framework API, stay with Python base types as far as possible instead of introducing a number of new data structures. Let the framework derive meta information such as names, types and documentation for the operation, its inputs, and its outputs from the user's Python code. It shall be possible to register any Python-callable of the from `f(*args, **kwargs)` as an operation.

Add extra meta-information to operations

Description Initial operation meta-information will be derived from Python code introspection. It shall include the user function's docstring and information about the arguments an its return values, exploiting any type annotations. For example, the following properties can be associated with input arguments: data type, default value, value set, valid range, if it is mandatory or optional, expected dataset schema so that operations can be ECV-specific. Meta-information is required to let an operation explain itself when used in a (IPython) REPL or when web service is requested to respond with an operations's capabilities. API users shall be able to extend the initial meta-information derived from Python code.

URD-Source

- CCIT-UR-LM0006: offer default values for lower level users as well as selectable options for higher level users.
- CCIT-UR-LM0002: accommodating ECV-specific processors in cases where the processing is specific to an ECV.

Static annotation vs. dynamic, programmatic registration

Description Operation registration and meta-information extension shall also be done by operation class / function *decorators*. The API shall provide a simple set of dedicated decorators that API user's attach to their operations. They will automatically register the user function as operation and add any extra meta-information.

Operation monitoring

Description Operation registration should recognise an optional *monitor* argument of a user function: `f(*args, monitor=Monitor.NONE, **kwargs)`. In this case the a monitor (of type `Monitor`) will be passed by the framework to the user function in order to observe the progress and to cancel an operation.

9.2.3 Verification

The module's unit-tests are located in test/test_op.py and may be executed using `$ py.test test/test_op.py --cov=cate/core/plugin.py` for extra code coverage information.

9.2.4 Components

cate.core.op.**OP_REGISTRY = OP_REGISTRY**
The default operation registry of type *cate.core.op.OpRegistry*.

class cate.core.op.**OpRegistry**
An operation registry allows for addition, removal, and retrieval of operations.

add_op(*operation: Callable*, *fail_if_exists=True*, *replace_if_exists=False*) → cate.core.op.Operation
Add a new operation registration.

Return type *Operation*

Parameters

- **operation** (Callable) – A operation object such as a class or any callable.
- **fail_if_exists** (bool) – raise ValueError if the operation was already registered
- **replace_if_exists** (bool) – replaces an existing operation if *fail_if_exists* is False

Returns a new or existing *cate.core.op.Operation*

get_op(*operation*, *fail_if_not_exists=False*) → cate.core.op.Operation
Get an operation registration.

Return type *Operation*

Parameters

- **operation** – A fully qualified operation name or operation object such as a class or any callable.
- **fail_if_not_exists** (bool) – raise ValueError if no such operation was found

Returns a *cate.core.op.Operation* object or None if *fail_if_not_exists* is False.

get_op_key(*operation: Union[str, typing.Callable]*)
Get a key under which the given operation will be registered.

Parameters **operation** – A fully qualified operation name or a callable object

Returns The operation key

op_registrations
Get all operation registrations of type *cate.core.op.Operation*.

Returns a mapping of fully qualified operation names to operation registrations

remove_op(*operation: Callable*, *fail_if_not_exists=False*) → Union[cate.core.op.Operation, NoneType]
Remove an operation registration.

Parameters

- **operation** (Callable) – A fully qualified operation name or operation object such as a class or any callable.
- **fail_if_not_exists** (bool) – raise ValueError if no such operation was found

Returns the removed *cate.core.op.Operation* object or `None` if *fail_if_not_exists* is `False`.

class cate.core.op.**Operation**(*wrapped_op: Callable*, *op_meta_info=None*)

An Operation comprises a wrapped callable (e.g. function, constructor, lambda form) and additional meta-information about the wrapped operation itself and its inputs and outputs.

Parameters

- **wrapped_op** – some callable object that will be wrapped.
- **op_meta_info** – operation meta information.

op_meta_info

Returns Meta-information about the operation, see `cate.core.op.OpMetaInfo`.

wrapped_op

Returns The actual operation object which may be any callable.

cate.core.op.**new_expression_op**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *expression: str*) → cate.core.op.Operation

Create an operation that wraps a Python expression.

Return type *Operation*

Parameters

- **op_meta_info** (`OpMetaInfo`) – Meta-information about the resulting operation and the operation's inputs and outputs.
- **expression** (`str`) – The Python expression. May refer to any name given in *op_meta_info.input*.

Returns The Python expression wrapped into an operation.

cate.core.op.**new_subprocess_op**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *command_pattern: str*, *run_python: bool = False*, *cwd: Union[str, NoneType] = None*, *env: Dict[str, str] = None*, *shell: bool = False*, *started: Union[str, typing.Callable] = None*, *progress: Union[str, typing.Callable] = None*, *done: Union[str, typing.Callable] = None*) → cate.core.op.Operation

Create an operation for a child program run in a new process.

Return type *Operation*

Parameters

- **op_meta_info** (`OpMetaInfo`) – Meta-information about the resulting operation and the operation's inputs and outputs.
- **command_pattern** (`str`) – A pattern that will be interpolated to obtain the actual command to be executed. May contain "{input_name}" fields which will be replaced by the actual input value converted to text. *input_name* must refer to a valid operation input name in *op_meta_info.input* or it must be the value of either the "write_to" or "read_from" property of another input's property map.
- **run_python** (`bool`) – If True, *command_pattern* refers to a Python script which will be executed with the Python interpreter that Cate uses.
- **cwd** – Current working directory to run the command line in.
- **env** (`Dict`) – Environment variables passed to the shell that executes the command line.
- **shell** (`bool`) – Whether to use the shell as the program to execute.

- **started** – Either a callable that receives a text line from the executable's stdout and returns a tuple (label, total_work) or a regex that must match in order to signal the start of progress monitoring. The regex must provide the group names "label" or "total_work" or both, e.g. "(?P<label>w+)" or "(?P<total_work>d+)"
- **progress** – Either a callable that receives a text line from the executable's stdout and returns a tuple (work, msg) or a regex that must match in order to signal process. The regex must provide group names "work" or "msg" or both, e.g. "(?P<msg>w+)" or "(?P<work>d+)"
- **done** – Either a callable that receives a text line a text line from the executable's stdout and returns True or False or a regex that must match in order to signal the end of progress monitoring.

Returns The executable wrapped into an operation.

cate.core.op.**op**(*tags=UNDEFINED*, *version=UNDEFINED*, *res_pattern=UNDEFINED*, *deprecated=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

`op` is a decorator function that registers a Python function or class in the default operation registry or the one given by *registry*, if any. Any other keywords arguments in *header* are added to the operation's meta-information header. Classes annotated by this decorator must have callable instances.

When a function is registered, an introspection is performed. During this process, initial operation the meta-information header property *description* is derived from the function's docstring.

If any output of this operation will have its history information automatically updated, there should be version information found in the operation header. Thus it's always a good idea to add it to all operations:

```
@op(version='X.x')
```

Parameters

- **tags** – An optional list of string tags.
- **version** – An optional version string.
- **res_pattern** – An optional pattern that will be used to generate the names for data resources that are used to hold a reference to the objects returned by the operation and that are cached in a Cate workspace. Currently, the only pattern variable that is supported and that must be present is `{index}` which will be replaced by an integer number that is guaranteed to produce a unique resource name.
- **deprecated** – An optional boolean or a string. If a string is used, it should explain why the operation has been deprecated and which new operation to use instead. If set to `True`, the operation's doc-string should explain the deprecation.
- **registry** – The operation registry.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of operation.

cate.core.op.**op_input**(*input_name: str*, *default_value=UNDEFINED*, *units=UNDEFINED*, *data_type=UNDEFINED*, *nullable=UNDEFINED*, *value_set_source=UNDEFINED*, *value_set=UNDEFINED*, *value_range=UNDEFINED*, *script_lang=UNDEFINED*, *deprecated=UNDEFINED*, *position=UNDEFINED*, *context=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

`op_input` is a decorator function that provides meta-information for an operation input identified by *input_name*. If the decorated function or class is not registered as an operation yet, it is added to the default operation registry or the one given by *registry*, if any.

When a function is registered, an introspection is performed. During this process, initial operation meta-information input properties are derived for each positional and keyword argument named *input_name*:

Derived property	Source
position	The position of a positional argument, e.g. `2` for input `z` in `def f(x, y, z, c=2).`
default_value	The value of a keyword argument, e.g. `52.3` for input `latitude` from argument definition `latitude:float=52.3`
data_type	The type annotation type, e.g. `float` for input `latitude` from argument definition `latitude:float`

The derived properties listed above plus any of *value_set*, *value_range*, and any key-value pairs in *properties* are added to the input's meta-information. A key-value pair in *properties* will always overwrite the derived properties listed above.

Parameters

- **`input_name`** (`str`) – The name of an input.
- **`default_value`** – A default value.
- **`units`** – The geo-physical units of the input value.
- **`data_type`** – The data type of the input values. If not given, the type of any given, non-None *default_value* is used.
- **`nullable`** – If `True`, the value of the input may be `None`. If not given, it will be set to `True` if the *default_value* is `None`.
- **`value_set_source`** – The name of an input, which can be used to generate a dynamic value set.
- **`value_set`** – A sequence of the valid values. Note that all values in this sequence must be compatible with *data_type*.
- **`value_range`** – A sequence specifying the possible range of valid values.
- **`script_lang`** – The programming language for a parameter of data_type "str" that provides source code of a script, e.g. "python".
- **`deprecated`** – An optional boolean or a string. If a string is used, it should explain why the input has been deprecated and which new input to use instead. If set to `True`, the input's doc-string should explain the deprecation.
- **`position`** – The zero-based position of an input.
- **`context`** – If `True`, the value of the operation input will be a dictionary representing the current execution context. For example, when the operation is executed from a workflow, the dictionary will hold at least three entries: `workflow` provides the current workflow, `step` is the currently executed step, and `value_cache` which is a mapping from step identifiers to step outputs. If *context* is a string, the value of the operation input will be the result of evaluating the string as Python expression with the current execution context as local environment. This means, *context* may be an expression such as 'value_cache', 'workspace.base_dir', 'step', 'step.id'.
- **`properties`** – Other properties (keyword arguments) that will be added to the meta-information of the named output.
- **`registry`** – Optional operation registry.

cate.core.op.**op_output**(*output_name: str*, *data_type=UNDEFINED*, *deprecated=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

op_output is a decorator function that provides meta-information for an operation output identified by *output_name*. If the decorated function or class is not registered as an operation yet, it is added to the default operation registry or the one given by *registry*, if any.

If your function does not return multiple named outputs, use the *op_return()* decorator function. Note that:

```
@op_return(...)
def my_func(...):
    ...
```

if equivalent to:

```
@op_output('return', ...)
def my_func(...):
    ...
```

To automatically add information about cate, its version, this operation and its inputs, to this output, set 'add_history' to True:

```
@op_output('name', add_history=True)
```

Note that the operation should have version information added to it when add_history is True:

```
@op(version='X.x')
```

Parameters

- **output_name** (str) – The name of the output.
- **data_type** – The data type of the output value.
- **deprecated** – An optional boolean or a string. If a string is used, it should explain why the output has been deprecated and which new output to use instead. If set to True, the output's doc-string should explain the deprecation.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of the named output.
- **registry** – Optional operation registry.

cate.core.op.**op_return**(*data_type=UNDEFINED*, *registry=OP_REGISTRY*, ***properties*)

op_return is a decorator function that provides meta-information for a single, anonymous operation return value (whose output name is "return"). If the decorated function or class is not registered as an operation yet, it is added to the default operation registry or the one given by *registry*, if any. Any other keywords arguments in *properties* are added to the output's meta-information.

When a function is registered, an introspection is performed. During this process, initial operation meta-information output properties are derived from the function's return type annotation, that is *data_type* will be e.g. float if a function is annotated as `def f(x, y) -> float: ...`.

The derived *data_type* property and any key-value pairs in *properties* are added to the output's meta-information. A key-value pair in *properties* will always overwrite a derived *data_type*.

If your function returns multiple named outputs, use the *op_output()* decorator function. Note that:

```
@op_return(...)
def my_func(...):
    ...
```

if equivalent to:

```
@op_output('return', ...)
def my_func(...):
    ...
```

To automatically add information about cate, its version, this operation and its inputs, to this output, set 'add_history' to True:

```
@op_return(add_history=True)
```

Note that the operation should have version information added to it when add_history is True:

```
@op(version='X.x')
```

Parameters

- **data_type** – The data type of the return value.
- **properties** – Other properties (keyword arguments) that will be added to the meta-information of the return value.
- **registry** – The operation registry.

9.3 Module cate.core.workflow

9.3.1 Description

Provides classes that are used to construct processing *workflows* (networks, directed acyclic graphs) from processing *steps* including Python callables, Python expressions, external processes, and other workflows.

This module provides the following data types:

- A *Node* has zero or more *inputs* and zero or more *outputs* and can be invoked
- A *Workflow* is a Node that is composed of Step objects
- A *Step* is a Node that is part of a Workflow and performs some kind of data processing.
- A *OpStep* is a Step that invokes a Python operation (any callable).
- A *ExpressionStep* is a Step that executes a Python expression string.
- A *WorkflowStep* is a Step that executes a Workflow loaded from an external (JSON) resource.
- A *NodePort* belongs to exactly one Node. Node ports represent both the named inputs and outputs of node. A node port has a name, a property source, and a property value. If source is set, it must be another NodePort that provides the actual port's value. The value of the value property can be basically anything that has an external (JSON) representation.

Workflow input ports are usually unspecified, but value may be set. Workflow output ports and a step's input ports are usually connected with output ports of other contained steps or inputs of the workflow via the source attribute. A step's output ports are usually unconnected because their value attribute is set by a step's concrete implementation.

Step node inputs and workflow outputs are indicated in the input specification of a node's external JSON representation:

- {"source": "NODE_ID.PORT_NAME" }: the output (or input) named *PORT_NAME* of another node given by *NODE_ID*.

- `{"source": ".PORT_NAME" }`: current step's output (or input) named *PORT_NAME* or of any of its parents.
- `{"source": "NODE_ID" }`: the one and only output of a workflow or of one of its nodes given by *NODE_ID*.
- `{"value": NUM|STR|LIST|DICT|null }`: a constant (JSON) value.

Workflows are callable by the CLI in the same way as single operations. The command line form for calling an operation is currently::

```
cate run OP|WORKFLOW [ARGS]
```

Where *OP* is a registered operation and *WORKFLOW* is a JSON file containing a JSON workflow representation.

9.3.2 Technical Requirements

Combine processors and other operations to create operation chains or processing graphs

Description Provide the means to connect multiple processing steps, which may be registered operations, operating system calls, remote service invocations.

URD-Sources

- CCIT-UR-LM0001: processor management allowing easy selection of tools and functionalities.
- CCIT-UR-LM0003: easy construction of graphs without any knowledge of a programming language (Graph Builder).
- CCIT-UR-LM0004: selection of a number of predefined standard processing chains.
- CCIT-UR-LM0005: means to configure a processor chain comprised of one processor only from the library to execute on data from the Common Data Model.

Integration of external, ECV-specific programs

Description Some processing step might only be solved by executing an external tool. Therefore, a special workflow step shall allow for invocation of external programs hereby mapping input values to program arguments, and program outputs to step outputs. It shall also be possible to monitor the state of the running sub-process.

URD-Source

- CCIT-UR-LM0002: accommodating ECV-specific processors in cases where the processing is specific to an ECV.

Programming language neutral representation

Description Processing graphs must be representable in a programming language neutral representation such as XML, JSON, YAML, so they can be designed by non-programmers and can be easily serialised, e.g. for communication with a web service.

URD-Source

- CCIT-UR-LM0003: easy construction of graphs without any knowledge of a programming language
- CCIT-UR-CL0001: reading and executing script files written in XML or similar

---

9.3.3 Verification

The module's unit-tests are located in test/test_workflow.py and may be executed using `$ py.test test/test_workflow.py --cov=cate/core/workflow.py` for extra code coverage information.

9.3.4 Components

class cate.core.workflow.**ExpressionStep**(*expression: str*, *inputs=None*, *outputs=None*, *node_id=None*)

An `ExpressionStep` is a step node that computes its output from a simple (Python) *expression* string.

Parameters

- **expression** – A simple (Python) expression string.
- **inputs** – input name to input properties mapping.
- **outputs** – output name to output properties mapping.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)

Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)

Create a new step node instance from the given *json_dict*

class cate.core.workflow.**NoOpStep**(*inputs: dict = None*, *outputs: dict = None*, *node_id: str = None*)

A `NoOpStep` "performs" a no-op, which basically means, it does nothing. However, it might still be useful to define step that or duplicates or renames output values by connecting its own output ports with any of its own input ports. In other cases it might be useful to have a `NoOpStep` as a placeholder or blackbox for some other real operation that will be put into place at a later point in time.

Parameters

- **inputs** – input name to input properties mapping.
- **outputs** – output name to output properties mapping.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)

Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)

Create a new step node instance from the given *json_dict*

class cate.core.workflow.**Node**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *node_id: str = None*)

Base class for all nodes including parent nodes (e.g. *Workflow*) and child nodes (e.g. *Step*).

All nodes have inputs and outputs, and can be invoked to perform some operation.

Inputs and outputs are exposed as attributes of the `input` and `output` properties and are both of type *NodePort*.

Parameters **node_id** – A node ID. If None, a name will be generated.

call(*context: Dict = None*, *monitor=Monitor.NONE*, *input_values: Dict = None*)

Calls this workflow with given *input_values* and returns the result.

The method does the following: 1. Set default_value where input values are missing in *input_values* 2. Validate the input_values using this workflows's meta-info 3. Set this workflow's input port values 4.

Invoke this workflow with given *context* and *monitor* 5. Get this workflow's output port values. Named outputs will be returned as dictionary.

Parameters

- **context** (Dict) – An optional execution context. It will be used to automatically set the value of any node input which has a "context" property set to either True or a context expression string.
- **monitor** – An optional progress monitor.
- **input_values** (Dict) – The input values.

Returns The output values.

collect_predecessors(*predecessors: List[_ForwardRef('Node')], excludes: List[_ForwardRef('Node')] = None*)
Collect this node (self) and preceding nodes in *predecessors*.

find_node(*node_id*) → Union[_ForwardRef('Node'), NoneType]
Find a (child) node with the given *node_id*.

find_port(*name*) → Union[_ForwardRef('NodePort'), NoneType]
Find port with given name. Output ports are searched first, then input ports. :param name: The port name :return: The port, or None if it couldn't be found.

id
The node's identifier.

inputs
The node's inputs.

invoke(*context: Dict = None*, *monitor: cate.util.monitor.Monitor = Monitor.NONE*) → None
Invoke this node's underlying operation with input values from input. Output values in output will be set from the underlying operation's return value(s).

Parameters

- **context** (Dict) – An optional execution context.
- **monitor** (Monitor) – An optional progress monitor.

max_distance_to(*other_node: cate.core.workflow.Node*) → int
If *other_node* is a source of this node, then return the number of connections from this node to *node*. If it is a direct source return 1, if it is a source of the source of this node return 2, etc. If *other_node* is this node, return 0. If *other_node* is not a source of this node, return -1.

Return type int

Parameters **other_node** – The other node.

Returns The distance to *other_node*

op_meta_info
The node's operation meta-information.

outputs
The node's outputs.

parent_node
The node's parent node or None if this node has no parent.

requires(*other_node: cate.core.workflow.Node*) → bool
Does this node require *other_node* for its computation? Is *other_node* a source of this node?

Return type bool

Parameters **other_node** – The other node.

Returns `True` if this node is a target of *other_node*

root_node
The root_node node.

set_id(*node_id: str*) → None
Set the node's identifier.

Parameters **node_id** (`str`) – The new node identifier. Must be unique within a workflow.

to_json_dict()
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

update_sources()
Resolve unresolved source references in inputs and outputs.

update_sources_node_id(*changed_node: cate.core.workflow.Node*, *old_id: str*)
Update the source references of input and output ports from *old_id* to *new_id*.

class `cate.core.workflow.`**NodePort**(*node: cate.core.workflow.Node*, *name: str*)
Represents a named input or output port of a *Node*.

to_json(*force_dict=False*)
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

update_source()
Resolve this node port's source reference, if any.

If the source reference has the form *node-id.port-name* then *node-id* must be the ID of the workflow or any contained step and *port-name* must be a name either of one of its input or output ports.

If the source reference has the form *.port-name* then *node-id* will refer to either the current step or any of its parent nodes that contains an input or output named *port-name*.

If the source reference has the form *node-id* then *node-id* must be the ID of the workflow or any contained step which has exactly one output.

If *node-id* refers to a workflow, then *port-name* is resolved first against the workflow's inputs followed by its outputs. If *node-id* refers to a workflow's step, then *port-name* is resolved first against the step's outputs followed by its inputs.

Raises **ValueError** – if the source reference is invalid.

update_source_node_id(*node: cate.core.workflow.Node*, *old_node_id: str*) → None
A node identifier has changed so we update the source references and clear the source of input and output ports from *old_node_id* to *node.id*.

Parameters

- **node** (*Node*) – The node whose identifier changed.
- **old_node_id** (`str`) – The former node identifier.

class `cate.core.workflow.`**OpStep**(*operation*, *node_id: str = None*, *registry=OP_REGISTRY*)
An *OpStep* is a step node that invokes a registered operation of type `Operation`.

Parameters

- **operation** – A fully qualified operation name or operation object such as a class or callable.

- **registry** – An operation registry to be used to lookup the operation, if given by name.
- **node_id** – A node ID. If None, a unique ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)
Create a new step node instance from the given *json_dict*

class cate.core.workflow.**OpStepBase**(*op: cate.core.op.Operation*, *node_id: str = None*)
Base class for concrete steps based on an `Operation`.

Parameters

- **op** – An `Operation` object.
- **node_id** – A node ID. If None, a unique ID will be generated.

op
The operation registration. See *cate.core.op.Operation*

class cate.core.workflow.**SourceRef**(*node_id*, *port_name*)

node_id
Alias for field number 0

port_name
Alias for field number 1

class cate.core.workflow.**Step**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *node_id: str = None*)
A step is an inner node of a workflow.

Parameters **node_id** – A node ID. If None, a name will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*) → Union[_ForwardRef('Step'), NoneType]
Create a new step node instance from the given *json_dict*

parent_node
The node's ID.

persistent
Return whether this step is persistent. That is, if the current workspace is saved, the result(s) of a persistent step may be written to a "resource" file in the workspace directory using this step's ID as filename. The file format and filename extension will be chosen according to each result's data type. On next attempt to execute the step again, e.g. if a workspace is opened, persistent steps may read the "resource" file to produce the result rather than performing an expensive re-computation. :return: True, if so, False otherwise

to_json_dict()
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

class cate.core.workflow.**SubProcessStep**(*command: str*, *run_python: bool = False*, *env: Dict[str, str] = None*, *cwd: str = None*, *shell: bool = False*, *started_re: str = None*, *progress_re: str = None*, *done_re: str = None*, *inputs: Dict[str, Dict] = None*, *outputs: Dict[str, Dict] = None*, *node_id: str = None*)

A `SubProcessStep` is a step node that computes its output by a sub-process created from the given *program*.

Parameters

- **command** – A pattern that will be interpolated by input values to obtain the actual command (program with arguments) to be executed. May contain "{input_name}" fields which will be replaced by the actual input value converted to text. *input_name* must refer to a valid operation input name in *op_meta_info.input* or it must be the value of either the "write_to" or "read_from" property of another input's property map.
- **run_python** – If True, *command_line_pattern* refers to a Python script which will be executed with the Python interpreter that Cate uses.
- **cwd** – Current working directory to run the command line in.
- **env** – Environment variables passed to the shell that executes the command line.
- **shell** – Whether to use the shell as the program to execute.
- **started_re** – A regex that must match a text line from the process' stdout in order to signal the start of progress monitoring. The regex must provide the group names "label" or "total_work" or both, e.g. "(?P<label>w+)" or "(?P<total_work>d+)"
- **progress_re** – A regex that must match a text line from the process' stdout in order to signal process. The regex must provide group names "work" or "msg" or both, e.g. "(?P<msg>w+)" or "(?P<work>d+)"
- **done_re** – A regex that must match a text line from the process' stdout in order to signal the end of progress monitoring.
- **inputs** – input name to input properties mapping.
- **outputs** – output name to output properties mapping.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)
Create a new step node instance from the given *json_dict*

class cate.core.workflow.**ValueCache**
`ValueCache` is a closable dictionary that maintains unique IDs for it's keys. If a `ValueCache` is closed, all closable values are also closed. A value is closeable if it has a `close` attribute whose value is a callable.

child(*key: str*) → cate.core.workflow.ValueCache
Return the child `ValueCache` for given *key*.

clear() → None
Override the `dict` method to closes values and remove all IDs.

close() → None
Close all values and remove all IDs.

get_id(*key: str*)
Return the integer ID for given *key* or `None`.

get_key(*id: int*)
Return the key for given integer *id* or `None`.

get_update_count(*key: str*)
Return the integer update count for given *key* or `None`.

get_value_by_id(*id: int*, *default=UNDEFINED*)
Return the value for the given integer *id* or return *default*.

pop(*key*, *default=None*)
Override the `dict` method to close the value and remove its ID.

rename_key(*key: str*, *new_key: str*) → None
Rename the given *key* into *new_key* without changing the value of the ID.

Parameters

- **key** (`str`) – The old key.
- **new_key** (`str`) – The new key.

`cate.core.workflow.`**WORKFLOW_SCHEMA_VERSION = 1**
Version number of Workflow JSON schema. Will be incremented with the first schema change after public release.

class `cate.core.workflow.`**Workflow**(*op_meta_info: cate.util.opmetainf.OpMetaInfo*, *node_id: str = None*)
A workflow of (connected) steps.

Parameters

- **op_meta_info** – Meta-information object of type `OpMetaInfo`.
- **node_id** – A node ID. If None, an ID will be generated.

find_node(*step_id: str*) → Union[_ForwardRef('Step'), NoneType]
Find a (child) node with the given *node_id*.

find_steps_to_compute(*step_id: str*) → List[_ForwardRef('Step')]
Compute the list of steps required to compute the output of the step with the given *step_id*. The order of the returned list is its execution order, with the step given by *step_id* is the last one.

Return type `List`

Parameters **step_id** (`str`) – The step to be computed last and whose output value is requested.

Returns a list of steps, which is never empty

invoke_steps(*steps: List[_ForwardRef('Step')]*, *context: Dict = None*, *monitor_label: str = None*, *monitor=Monitor.NONE*) → None
Invoke just the given steps.

Parameters

- **steps** (`List`) – Selected steps of this workflow.
- **context** (`Dict`) – An optional execution context
- **monitor_label** (`str`) – An optional label for the progress monitor.
- **monitor** – The progress monitor.

classmethod load(*file_path_or_fp: Union[str, io.IOBase]*, *registry=OP_REGISTRY*) → cate.core.workflow.Workflow
Load a workflow from a file or file pointer. The format is expected to be "Workflow JSON".

Parameters

- **file_path_or_fp** – file path or file pointer
- **registry** – Operation registry

Returns a workflow

remove_orphaned_sources(*removed_node: cate.core.workflow.Node*)
Remove all input/output ports, whose source is still referring to *removed_node*. :type removed_node: *Node* :param removed_node: A removed node.

classmethod sort_steps(*steps: List[_ForwardRef('Step')]*)
Sorts the list of workflow steps in the order they they can be executed.

sorted_steps
The workflow steps in the order they they can be executed.

steps
The workflow steps in the order they where added.

store(*file_path_or_fp: Union[str, io.IOBase]*) → None
Store a workflow to a file or file pointer. The format is "Workflow JSON".

Parameters **file_path_or_fp** – file path or file pointer

to_json_dict() → dict
Return a JSON-serializable dictionary representation of this object.

Returns A JSON-serializable dictionary

update_sources() → None
Resolve unresolved source references in inputs and outputs.

update_sources_node_id(*changed_node: cate.core.workflow.Node*, *old_id: str*)
Update the source references of input and output ports from *old_id* to *new_id*.

class cate.core.workflow.**WorkflowStep**(*workflow: cate.core.workflow.Workflow*, *resource: str*, *node_id: str = None*)
A *WorkflowStep* is a step node that invokes an externally stored *Workflow*.

Parameters

- **workflow** – The referenced workflow.
- **resource** – A resource (e.g. file path, URL) from which the workflow was loaded.
- **node_id** – A node ID. If None, an ID will be generated.

enhance_json_dict(*node_dict: collections.OrderedDict*)
Enhance the given JSON-compatible *node_dict* by step specific elements.

classmethod new_step_from_json_dict(*json_dict*, *registry=OP_REGISTRY*)
Create a new step node instance from the given *json_dict*

resource
The workflow's resource path (file path, URL).

workflow
The workflow.

cate.core.workflow.**new_workflow_op**(*workflow_or_path: Union[str, cate.core.workflow.Workflow]*) → cate.core.op.Operation
Create an operation from a workflow read from the given path.

Return type *Operation*

Parameters **workflow_or_path** – Either a path to Workflow JSON file or *Workflow* object.

Returns The workflow operation.

9.4 Module cate.core.plugin

9.4.1 Description

The cate.core.plugin module exposes the Cate's plugin REGISTRY which is mapping from Cate entry point names to plugin meta information. An Cate plugin is any callable in an internal/extension module registered with cate_plugins entry point.

Clients register a Cate plugin in the setup() call of their setup.py script. The following plugin example comprises a main module cate_wavelet_gapfill which provides the entry point function cate_init::

```
setup(
    name="cate-gapfill-wavelet",
    version="0.5",
    description='A wavelet-based gap-filling algorithm for the ESA CCI Toolbox',
    license='GPL 3',
    author='John Doe',
    packages=['cate_wavelet_gapfill'],
    entry_points={
        'cate_plugins': [
            'cate_wavelet_gapfill = cate_wavelet_gapfill:cate_init',
        ],
    },
    install_requires=['pywavelets >= 2.1'],
)
```

The entry point callable should have the following signature:

```
def cate_init(*args, **kwargs):
    pass
```

or:

```
class EctInit:
    def __init__(*args, **kwargs)__:
        pass
```

The return values are ignored.

9.4.2 Verification

The module's unit-tests are located in test/test_plugin.py and may be executed using $ py.test test/test_plugin.py --cov=cate/core/plugin.py for extra code coverage information.

9.4.3 Components

cate.core.plugin.**PLUGIN_REGISTRY = {'cate_ds': {'entry_point': 'cate_ds'}, 'cate_ops':**
Mapping of Cate entry point names to JSON-serializable plugin meta-information.

cate.core.plugin.**cate_init**(**arg*, ***kwargs*)
No actual use, just demonstrates the signature of an Cate entry point callable.

Parameters

- **arg** – any arguments (not used)

- **kwargs** – any keyword arguments (not used)

Returns any or void (not used)

9.5 Module cate.conf

9.6 Module cate.ds

9.6.1 Description

The ds package comprises all specific data source implementations.

This is a plugin package automatically imported by the installation script's entry point cate_ds (see the projects setup.py file).

9.6.2 Verification

The module's unit-tests are located in test/ds and may be executed using $ py.test test/ops/test_<MODULE>.py --cov=cate/ops/<MODULE>.py for extra code coverage information.

9.6.3 Components

9.7 Module cate.ops

9.7.1 Description

The ops package comprises all specific operation and processor implementations.

This is a plugin package automatically imported by the installation script's entry point cate_ops (see the projects setup.py file).

9.7.2 Verification

The module's unit-tests are located in test/ops and may be executed using $ py.test test/ops/test_<MODULE>.py --cov=cate/ops/<MODULE>.py for extra code coverage information.

9.7.3 Functions

cate.ops.**resample_2d**(*src*, *w*, *h*, *ds_method=54*, *us_method=11*, *fill_value=None*, *mode_rank=1*, *out=None*)

Resample a 2-D grid to a new resolution.

Parameters

- **src** – 2-D *ndarray*
- **w** – *int* New grid width
- **h** – *int* New grid height

- **ds_method** (int) – one of the *DS_* constants, optional Grid cell aggregation method for a possible downsampling
- **us_method** (int) – one of the *US_* constants, optional Grid cell interpolation method for a possible upsampling
- **fill_value** – *scalar*, optional If None, it is taken from **src** if it is a masked array, otherwise from *out* if it is a masked array, otherwise numpy's default value is used.
- **mode_rank** (int) – *scalar*, optional The rank of the frequency determined by the *ds_method* DS_MODE. One (the default) means most frequent value, zwo means second most frequent value, and so forth.
- **out** – 2-D *ndarray*, optional Alternate output array in which to place the result. The default is *None*; if provided, it must have the same shape as the expected output.

Returns An resampled version of the *src* array.

cate.ops.**downsample_2d**(*src*, *w*, *h*, *method=54*, *fill_value=None*, *mode_rank=1*, *out=None*)

Downsample a 2-D grid to a lower resolution by aggregating original grid cells.

Parameters

- **src** – 2-D *ndarray*
- **w** – *int* Grid width, which must be less than or equal to *src.shape[-1]*
- **h** – *int* Grid height, which must be less than or equal to *src.shape[-2]*
- **method** (int) – one of the *DS_* constants, optional Grid cell aggregation method
- **fill_value** – *scalar*, optional If None, it is taken from **src** if it is a masked array, otherwise from *out* if it is a masked array, otherwise numpy's default value is used.
- **mode_rank** (int) – *scalar*, optional The rank of the frequency determined by the *method* DS_MODE. One (the default) means most frequent value, zwo means second most frequent value, and so forth.
- **out** – 2-D *ndarray*, optional Alternate output array in which to place the result. The default is *None*; if provided, it must have the same shape as the expected output.

Returns A downsampled version of the *src* array.

cate.ops.**upsample_2d**(*src*, *w*, *h*, *method=11*, *fill_value=None*, *out=None*)

Upsample a 2-D grid to a higher resolution by interpolating original grid cells.

Parameters

- **src** – 2-D *ndarray*
- **w** – *int* Grid width, which must be greater than or equal to *src.shape[-1]*
- **h** – *int* Grid height, which must be greater than or equal to *src.shape[-2]*
- **method** (int) – one of the *US_* constants, optional Grid cell interpolation method
- **fill_value** – *scalar*, optional If None, it is taken from **src** if it is a masked array, otherwise from *out* if it is a masked array, otherwise numpy's default value is used.
- **out** – 2-D *ndarray*, optional Alternate output array in which to place the result. The default is *None*; if provided, it must have the same shape as the expected output.

Returns An upsampled version of the *src* array.

9.8 Module `cate.cli.main`

9.8.1 Description

This module provides Cate's CLI executable.

To use the CLI executable, invoke the module file as a script, type `python3 cate/cli/main.py [ARGS] [OPTIONS]`. Type *python3 cate/cli/main.py –help'* for usage help.

The CLI operates on sub-commands. New sub-commands can be added by inheriting from the `Command` class and extending the `Command.REGISTRY` list of known command classes.

9.8.2 Technical Requirements

Extensible CLI with multiple sub-commands

Description The CCI Toolbox should only have a single CLI executable that comes with multiple sub-commands instead of maintaining a number of different executables for each purpose. Plugins shall be able to add new CLI sub-commands.

URD-Source

- CCIT-UR-CR0001: Extensibility.
- CCIT-UR-A0002: Offer a Command Line Interface (CLI).

Run operations and workflows

Description Allow for executing registered operations an workflows composed of operations.

URD-Source

- CCIT-UR-CL0001: Reading and executing script files written in XML or similar

---

List available data, operations and extensions

Description Allow for listing dynamic content including available data, operations and plugin extensions.

URD-Source

- CCIT-UR-E0001: Dynamic extension by the use of plug-ins

Display information about available climate data sources

Description Before downloading ECV datasets to the local computer, users shall be able to display information about them, e.g. included variables, total size, spatial and temporal resolution.

URD-Source

- CCIT-UR-DM0009: Holding information of any CCI ECV type
- CCIT-UR-DM0010: Attain meta-level status information per ECV type

---

Synchronize locally cached climate data

Description Allow for listing dynamic content including available data, operations and plugin extensions.

URD-Source

- CCIT-UR-DM0006: Access to and ingestion of ESA CCI datasets

---

9.8.3 Verification

The module's unit-tests are located in test/cli/test_main.py and may be executed using `$ py.test test/cli/test_main.py --cov=cate/cli/test_main.py` for extra code coverage information.

9.8.4 Components

`cate.cli.main.`**`CLI_NAME = 'cate'`**
Name of the Cate CLI executable (= `cate`).

`cate.cli.main.`**`COMMAND_REGISTRY = [<class 'cate.cli.main.DataSourceCommand'>, <class 'cate.c`**
List of sub-commands supported by the CLI. Entries are classes derived from `Command` class. Cate plugins may extend this list by their commands during plugin initialisation.

class `cate.cli.main.`**DataSourceCommand**
The `ds` command implements various operations w.r.t. datasets.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class `cate.cli.main.`**IOCommand**
The `io` command implements various operations w.r.t. supported data and file formats.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class cate.cli.main.**OperationCommand**
The `op` command implements various operations w.r.t. *operations*.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class cate.cli.main.**PluginCommand**
The `pi` command lists the content of various plugin registry.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class cate.cli.main.**ResourceCommand**
The `res` command implements various operations w.r.t. *workspaces*.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class cate.cli.main.**RunCommand**
The `run` command is used to invoke registered operations and JSON workflows.

classmethod configure_parser(*parser*)
Configure *parser*, i.e. make any required `parser.add_argument(*args, **kwargs)` calls. See https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.add_argument

Parameters **parser** – The command parser to configure.

execute(*command_args*)
Execute this command.

The command's arguments in *command_args* are attributes namespace returned by `argparse.ArgumentParser.parse_args()`. Also refer to to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.parse_args

`execute``implementations shall raise a ``CommandError` instance on failure.

Parameters **command_args** – The command's arguments.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class cate.cli.main.**UpdateCommand**
The `update` command is used to update an existing cate environment to a specific or the latest cate version.

classmethod configure_parser(*parser*)
Configure *parser*, i.e. make any required `parser.add_argument(*args, **kwargs)` calls. See https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.add_argument

Parameters **parser** – The command parser to configure.

execute(*command_args*)
Execute this command.

The command's arguments in *command_args* are attributes namespace returned by `argparse.ArgumentParser.parse_args()`. Also refer to to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.parse_args

`execute``implementations shall raise a ``CommandError` instance on failure.

Parameters **command_args** – The command's arguments.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

class cate.cli.main.**WorkspaceCommand**
The `ws` command implements various operations w.r.t. *workspaces*.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

classmethod name()

Returns A unique command name

classmethod parser_kwargs()
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

9.9 Module `cate.webapi`

9.10 Module `cate.util`

9.10.1 Description

The `cate.util` package provides application-independent utility functions.

This package is independent of other ‘‘cate.*‘‘packages and can therefore be used stand-alone.

9.10.2 Verification

The module's unit-tests are located in test/util and may be executed using `$ py.test test/util --cov=cate/util` for extra code coverage information.

9.10.3 Components

9.11 Module `cate.util.cache`

9.11.1 Description

This module defines the *Cache* class which represents a general-purpose cache. A cache is configured by a *CacheStore* which is responsible for storing and reloading cached items.

The default cache stores are

- *MemoryCacheStore*
- *FileCacheStore*

Every cache has capacity in physical units defined by the *CacheStore*. When the cache capacity is exceeded a replacement policy for cached items is applied until the cache size falls below a given ratio of the total capacity.

The default replacement policies are

- *POLICY_LRU*
- *POLICY_MRU*
- *POLICY_LFU*
- *POLICY_RR*

This package is independent of other ‘‘cate.*‘‘packages and can therefore be used stand-alone.

9.11.2 Components

class cate.util.cache.**Cache**(*store=<cate.util.cache.MemoryCacheStore object>*, *capacity=1000*, *threshold=0.75*, *policy=<function _policy_lru>*, *parent_cache=None*)

An implementation of a cache. See https://en.wikipedia.org/wiki/Cache_algorithms

class Item

Cache-private class representing an item in the cache.

class cate.util.cache.**CacheStore**

Represents a store to which cached values can be stored into and restored from.

can_load_from_key(*key*) → bool

Test whether a stored value representation can be loaded from the given key. :rtype: `bool` :param key: the key :return: True, if so

discard_value(*key*, *stored_value*)

Discard a value from it's storage. :param key: the key :param stored_value: the stored representation of the value

load_from_key(*key*)

Load a stored value representation of the value and its size from the given key. :param key: the key :return: a 2-element sequence containing the stored representation of the value and it's size

restore_value(*key*, *stored_value*)

Restore a vale from its stored representation. :param key: the key :param stored_value: the stored representation of the value :return: the item

store_value(*key*, *value*)

Store a value and return it's stored representation and size in any unit, e.g. in bytes. :param key: the key :param value: the value :return: a 2-element sequence containing the stored representation of the value and it's size

class cate.util.cache.**FileCacheStore**(*cache_dir: str*, *ext: str*)

Simple file store for values which can be written and read as bytes, e.g. encoded PNG images.

can_load_from_key(*key*) → bool

Test whether a stored value representation can be loaded from the given key. :rtype: `bool` :param key: the key :return: True, if so

discard_value(*key*, *stored_value*)

Discard a value from it's storage. :param key: the key :param stored_value: the stored representation of the value

load_from_key(*key*)

Load a stored value representation of the value and its size from the given key. :param key: the key :return: a 2-element sequence containing the stored representation of the value and it's size

restore_value(*key*, *stored_value*)

Restore a vale from its stored representation. :param key: the key :param stored_value: the stored representation of the value :return: the item

store_value(*key*, *value*)

Store a value and return it's stored representation and size in any unit, e.g. in bytes. :param key: the key :param value: the value :return: a 2-element sequence containing the stored representation of the value and it's size

class cate.util.cache.**MemoryCacheStore**

Simple memory store.

can_load_from_key(*key*) → bool
Test whether a stored value representation can be loaded from the given key. :rtype: `bool` :param key: the key :return: True, if so

discard_value(*key*, *stored_value*)
Clears the value in the given stored_value. :param key: the key :param stored_value: the stored representation of the value

load_from_key(*key*)
Load a stored value representation of the value and its size from the given key. :param key: the key :return: a 2-element sequence containing the stored representation of the value and it's size

restore_value(*key*, *stored_value*)

Parameters

- **key** – the key
- **stored_value** – the stored representation of the value

Returns the original value.

store_value(*key*, *value*)
Return (value, 1). :param key: the key :param value: the original value :return: the tuple (stored value, size) where stored value is the sequence [key, value].

cate.util.cache.**POLICY_LFU**(*item*)
Discard Least Frequently Used first

cate.util.cache.**POLICY_LRU**(*item*)
Discard Least Recently Used items first

cate.util.cache.**POLICY_MRU**(*item*)
Discard Most Recently Used first

cate.util.cache.**POLICY_RR**(*item*)
Discard items by Random Replacement

9.12 Module `cate.util.cli`

class cate.util.cli.**Command**
Represents a (sub-)command of a command-line interface.

classmethod configure_parser(*parser: argparse.ArgumentParser*) → None
Configure *parser*, i.e. make any required `parser.add_argument(*args, **kwargs)` calls. See https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.add_argument

Parameters **parser** (`ArgumentParser`) – The command parser to configure.

execute(*command_args: argparse.Namespace*) → None
Execute this command.

The command's arguments in *command_args* are attributes namespace returned by `argparse.ArgumentParser.parse_args()`. Also refer to to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.parse_args

`execute``implementations shall raise a ``CommandError` instance on failure.

Parameters **command_args** (`Namespace`) – The command's arguments.

classmethod name() → str

Returns A unique command name

classmethod new_monitor() → cate.util.monitor.Monitor
Create a new console progress monitor.

Returns a new Monitor instance.

classmethod parser_kwargs() → dict
Return parser keyword arguments dictionary passed to a `argparse.ArgumentParser(**parser_kwargs)` call.

For the possible keywords in the returned dictionary, refer to https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.

Returns A keyword arguments dictionary.

exception cate.util.cli.**CommandError**(*message*)
An error type signaling command-line errors.

Parameters **message** – Error message

class cate.util.cli.**NoExitArgumentParser**(**args*, ***kwargs*)
Special `argparse.ArgumentParser` that never directly exits the current process. It raises an `ExitException` instead.

exception ExitException(*status*, *message*)
Raises instead of exiting the current process.

exit(*status=0*, *message=None*)
Overrides the base class method in order to raise an `ExitException`.

class cate.util.cli.**SubCommandCommand**

classmethod configure_parser(*parser: argparse.ArgumentParser*) → None
Add a new sub-parsers to the given parser. Call `configure_parser_and_subparsers` with the new sub-parsers.

Parameters **parser** (`ArgumentParser`) – The command parser to configure.

classmethod configure_parser_and_subparsers(*parser*, *subparsers*)
Configure the given parser and its sub-parsers.

Overrides of this method must, e.g.:: list_parser = subparsers.add_parser('list', ...) # ... configure list_parser here, and finally set its "sub_command_function" like so: list_parser.set_defaults(sub_command_function=cls._execute_list)

Sub-command functions shall raise a `CommandError` instance on failure.

Parameters

- **parser** – The command parser to configure.
- **subparsers** – A factory for sub-command parsers.

execute(*command_args*)
Executes the function given by the "sub_command_function" attribute of given *command_args* with *command_args* as only argument.

Parameters **command_args** –

cate.util.cli.**run_main**(*name: str, description: str, version: str, command_classes: Sequence[cate.util.cli.Command], license_text: str = None, docs_url: str = None, error_message_trimmer=None, args: Sequence[str] = None*) → int
A CLI's entry point function.

To be used in your own code as follows:

```
>>> if __name__ == '__main__':
>>>     sys.exit(run_main(...))
```

Return type int

Parameters

- **name** (str) – The program's name.
- **description** (str) – The program's description.
- **version** (str) – The program's version string.
- **command_classes** (Sequence) – The CLI commands.
- **license_text** (str) – An optional license text.
- **docs_url** (str) – An optional documentation URL.
- **error_message_trimmer** – An optional callable (str)->str that trims error message strings.
- **args** (Sequence) – list of command-line arguments. If not passed, sys.argv[1:] is used.

Returns An exit code where 0 stands for success.

9.13 Module cate.util.im

9.13.1 Description

The cate.util.im package provides application-independent utility functions for working with tiled image pyramids.

The Cate project uses this package for implementing a RESTful web service that provides image tiles from image pyramids.

This package is independent of other cate.* packages, but it depends on the following external packages

- numpy
- pillow (for PIL)
- matplotlib

9.13.2 Verification

The module's unit-tests are located in test/util/im and may be executed using $ py.test test/util/im --cov=cate/util/im for extra code coverage information.

9.13.3 Components

9.14 Module cate.util.web

CHAPTER 10

Downloads

10.1 Installers

The Cate software can be downloaded from the Cate releases page under the terms and conditions of the MIT open source license.

We make available separate installers for *Cate* (Cate's Python environment and Python package `cate` comprising Cate API, Cate CLI, Cate WebAPI Service) and the *Cate Desktop* application (Cate's GUI).

Please check the [] if you encounter any problems installing Cate.

10.2 Source Code

The source code for Cate's Python core is available the in cate, Cate Desktop in the cate-desktop repository on GitHub. It is quite easy to build Cate from scratch and run it from the command-line.

CHAPTER 11

Support

11.1 User Forum

Please post any feedback, support requests and ideas for the future to the Cate User Forum.

11.2 Issue Tracking

Bugs, feature requests, suggestions for improvements should be reported in the Cate Issue Tracker.

11.3 Contributing

We are happy to receive pull requests from your fork of the Cate repository on GitHub.

11.4 Known Issues

11.4.1 Data Access

1. When running Cate / Cate Desktop on Windows and accessing data from the ESA Open Data Portal, you may receive a **SSL certificate verify failed** error. The workaround is to visit the ESGF Portal at CEDA web side using Edge, Chrome, or Firefox. This will cause your browser to register the website URL in question with your operating system's trusted SSL certificates. See also Issue #64.
2. Not all datasets from offered by the ESA CCI Open Data Portal can be used in Cate. Please check the ODP Datasets and Data Access Issues page to see whether you problem with a dataset is known and if there are already fixes / workarounds.

11.4.2 Other

We have collected all other known issues in the Cate Issue Tracker. If you encounter a problem, please search for it there first.

CHAPTER 12

Developer Guide

This chaper is aimed at people willing to contribute to Cate development, as well as a source of information and a reminder of best practices for people who already develop Cate.

12.1 Coding practices

12.1.1 Environment

We use Python 3.6+ and exploit its features. Developers are encouraged to use the Miniconda 64-bit environment. You can create a new development environment by using the `environment.yml` file located in the project root, type `conda env create --file environment.yml`. This will create a new dedicated Conda environment named `cate-env`.

(In order to generate Cate's documentation we use a simplified version of the environment `environment-rtd.yml` in the ReadTheDocs (RDT) configuration, see Docs on ReadTheDocs.)

Don't use any platform-specific features of Python!

12.1.2 Testing

- Write good tests. Good tests are ones that test expected core behaviour. Bad tests make it hard to refactor production code towards better architecture. See article Why Most Unit Testing is Waste. *Thanks, Ralf!*
- Target at 100% code coverage to make sure we don't access inexistent attributes, at least for a given test context. But remember that 100% code coverage does not imply 100% coverage of the possible configuration permutations (which can be close to infinity). Therefore it is still the quality of tests that provide value to the software and that result in high code coverage.
- Use `pytest` for testing, to run test with coverage type `pytest --cov=cate test`

12.1.3 Git

- Not push any code to `master` that isn't backed by one or more unit-tests.
- Keep `master` unbroken, only push if all test are green.
- Always create new branches for new experimental API or API revisions. Don't do that on `master`. Merge when branch is ready and reviewed and accepted by the team. Then delete your (remote) branch.
- When working in the official repository, there is a guideline for branch names. Use `issuenr-initials-description`.

12.2 Coding conventions

12.2.1 Code style

Like most Python projects, we try to adhere to PEP-8 (Style Guide for Python Code) and PEP-257 (Docstring Conventions) with any modifications documented here. Be sure to read those documents if you intend to contribute code to Cate project.

12.2.2 Spaces or tabs, etc

- According to PEP-8, we use 4 spaces for indention.
- Lines shall be no longer than 120 characters.
- Put a 2-line space between global classes, functions, variable declarations. Put a 1-line space between class methods.

12.2.3 Private components and properties

- According to PEP-8, we use a leading underscore to denote components as private.
- In most cases, class instance variables should be private. Use the `@property` annotation on a getter method to export them in a controlled way. Think twice if you want write access.

12.2.4 Docstrings

- Use triple quotes for docstrings `"""`.
- Use single docstrings `"""bla bla bla."""` if you have no docstring attributes and if the text fits into one line. Otherwise, add a line break after the opening `"""` and before the closing `"""`.
- Use the Sphinx-style docstring attributes, e.g. `:param <name>:`, `:return:`, etc and references, e.g. `:py:class:` or `py:meth:` etc.
- Use the Sphinx `#:` syntax to document variables.
- All modules must have a docstring that explains a module's intend, content, usage, and requirements that lead to its design.
- All public (API) classes must have a docstring that explains a class' (single!) purpose and its constructor parameters passed to the `__init__` method.

- All public (API) functions or methods must have a docstring that explains a function's or method's (single!) behaviour, parameter values + types and return value + type (if any).
- All public (API) variables must have a docstring that explains a variable's purpose, value and type.

See also PEP-257 and PEP-258.

12.2.5 Type annotations

Use type annotations when it makes sense. It makes sense, when it helps the IDE to point you to coding errors. It makes sense to help other people understand our code. When it makes sense, use type types from the new Python 3.5 `typing` module. However, if you allow a certain function argument to be of multiple types, don't try to construct wild type annotation expressions, because this will reduce the readability of the code again. In this case it is better to provide a reasonable docstring.

12.2.6 TODO comments

Feel free to use TODO comments in the code on your personal branches, but avoid them on `master`. If you need one, use following format

```
# TODO (forman, 20160613): bla bla bla
```

Ideally, TODO comments are backed by a GitHub issue providing more background info

```
# TODO (forman, 20160613): bla bla bla, see https://github.com/CCI-Tools/cate/issues/
↪39
```

12.3 API Usage

12.4 Plugin Development

12.5 Operation Development

In general, operation development follows the general Cate plugin development approach. E.g., any valid Python function can be decorated accordingly to introduce it to the Cate plugin system. However, there are some caveats one should keep in mind, as well as best practices to follow when developing Cate operations. This chapters explores these issues.

12.5.1 Operation development technology stack

To develop operations for cate one should be at least cursory familiar with the following Python projects:

- xarray
- numpy
- pandas
- dask

The `xarray` package is used the most as `xarray.Dataset` is the data model used to represent raster data throughout Cate. Most operations work on xarray datasets and produce xarray datasets. For tabular data representation and manipulation Cate supports `pandas`. `Numpy` is the corner stone of both xarray and pandas and is used when data is explicitly loaded into memory from an xarray object.

The `dask` package provides numpy-like data array abstraction of datasets spanning many on-disk files or even remote locations. A dask array is the underlying array object type of xarray datasets spanning over multiple files, which is the case in the large majority of Cate use cases. It can be beneficial to be accustomed with how dask works in order to write fast, parallelized operations. Not taking into account how dask works can result in a heavy performance penalty.

12.5.2 Registration with the Cate plugin system

Any python function can be registered in the cate operation registry by decorating it accordingly. As the bare minimum the `@op` decorator must be used. Depending on particular circumstances it may be needed to also use other decorators, such as `@op_input`, `@op_output`, `@op_return`.

For in-depth information on these decorators and their parameters, please check detailed design of *Module cate.core.op* as well as documentation on *Operation Management* and *Plugin Concept*.

A minimal Cate operation would then look like the following:

```
from cate.core.op import op


@op()
def dummy_operation(a, b):
    return a + b
```

A more involved example using tags to ease operation discovery by the user, as well as accepting file inputs, inputs consisting of known value sets, as well as variable inputs tied to a particular dataset would look like the following:

```
from cate.core.op import op, op_input
from cate.core.types import VarName


@op(tags=['geometric'])
@op_input('file', file_open_mode='w', file_filters=[dict(name='NetCDF', extensions=[
↪'nc'])])
@op_input('set', value_set=['a', 'b', 'c'])
@op_input('var', value_set_source='ds', data_type=VarName)
def some_operation(ds: xr.Dataset,
                   file: str,
                   set: str = 'a',
                   var: VarName.TYPE):
    # Do some science here
    return ds
```

In this example we have denoted input named `file` as an input that requires a file browser on the GUI, as well as inputs `set` and `var` as inputs that require a drop-down box on the GUI, as well as what values should be in this drop down box, or where to find them.

We also use the Cate typing system to let other parts of Cate (GUI, CLI) be aware of what the type of `var` is, as well as to enable streamlined validation. In light of operation development this is described in more detail here: *Cate typing system*.

If the newly created operation is meant to be part of the Cate core operation suite, it should be possible to import it when Cate is used programmatically. Hence, it should be put in `cate/ops` and imported in `cate/ops/__init__.py`.

Tags

Each operation should have at least one tag. This can be the module name, `input` or `output` in case of operations in the `io` module, as well as a tag from the following list:

- `utility` for any utility operations
- `internal` for internal operations, they will not be shown in user interfaces
- `geometric` for geometric operations
- `point` for operations that operate on single lon/lat points
- `spatial` for predominantly spatial operations
- `temporal` for predominantly temporal operations
- `filter` for operations that filter out things from an input to an output

Deprecations

Often it is required to change the name or the arguments of an existing operation. To provide backward compatibility the "old" operation can still be maintained by *deprecating* it. If we need to change a parameter name and possibly its type, we can also keep the old name and type and deprecate it.

The `deprecated` property is available for the `@op`, `@op_input`, and `@op_output` decorators. Its value may be just `True` or a string explaining why the operation/input/output has been deprecated and what to do instead.

```
@op(deprecated='"some_operation()" was inaccurate; use "some_new_operation()" instead
↪')
def some_operation(...):
    ...

@op_input('id', nullable=False)
@op_input('name', deprecated='Meaning changed; use "id" instead')
def some_operation(id:str = None, name:str = None, ...):
    id = id or name
    ...
```

Note, for maximum backward compatibility it is always a good idea to use keyword arguments instead of positional arguments.

By default, deprecated operations and deprecated inputs/outputs will not be be shown to users in the Cate CLI and Cate Desktop GUI.

To list all deprecated operations in the Cate CLI, type::

```
$ cate op list --deprecated
```

12.5.3 History information

Well behaved netCDF filters are expected to add information about themselves to the `history` attribute of a netCDF file. See Description of netCDF file contents.

Cate facilitates this by automatically adding information about Cate, the particular operation, it's version and invocation parameters to outputs that have been marked for history addition by providing the appropriate parameter to `@op_output` or `@op_return` decorators. Note that version information must be provided to the `@op` decorator as well.

```
from cate.core.op import op, op_output


@op(version='1.0')
@op_output('name2', add_history=True)
def my_op_that_saves_history_info(ds1: xr.Dataset, ds2: xr.Dataset):
    # Do some science
    return {'name1': ds1, 'name2': ds2}
```

Here history information will be added only to the `name2` outputs. We could have added `add_history=True` to both outputs. Adding history information to the only outputs, if this outputs is a dataset, can be achieved by using `@op_return` in a similar manner.

12.5.4 Cate typing system

Operations must use the Cate typing system to ensure that correct controls are shown in the GUI for the given inputs. Cate typing system also ensures that part of input validation can be done 'for free' and is located in the same place, as well as lets one create operations that mimic polymorphism by accepting multiple input types.

For example, an operation that accepts both an `xr.Dataset` and a `pd.DataFrame`, as well as takes a polygon could look like this:

```
from cate.core.types import DatasetLike, PolygonLike
from cate.core.op import op, op_input


@op()
@op_input('dsf', data_type=DatasetLike)
@op_input('region', data_type=PolygonLike)
def my_op_using_advanced_types(dsf: DatasetLike.TYPE, region: PolygonLike.TYPE):
    # Convert inputs to base types (implicit validation)
    ds = DatasetLike.convert(dsf)
    region = PolygonLike.convert(region)

    # Do some science

    return ds
```

Note that the framework requires that Cate typing system is used both in the decorator, as well as function signature. Here we have made an operation that accepts both `xr.Dataset` and a `pd.DataFrame` and converts it to an `xr.Dataset` for the actual calculation. We also have a `region` parameter that can be a `shapely.geometry.Polygon`, a coordinate string, a WKT string, a list of coordinate points, as well as a list of lon/lat values. Now the GUI is also aware that the operation expects a polygon and an appropriate dialog can be displayed.

12.5.5 Monitor usage

Operations that can be potentially long running should implement a Cate monitor that can be used by the CLI and the GUI to track the operation's progress, as well as to cancel the operation. It can sometimes be hard to determine whether a particular operation will be long running or not. In that case the rule of thumb should be to err on the side of implementing a monitor.

For example:

```
from cate.core.op import op
from cate.util.monitor import Monitor

@op()
def my_op_with_a_monitor(a: str, monitor: Monitor = Monitor.NONE):
    # Set up the monitor
    with monitor.starting('Monitor Operation', total_work=len(a)):
        for i in a:

            # Do work

            # Update the monitor
            monitor.progress(work=1)

            # If there are resources to clean up (e.g., open file handles)
            # use the following instead:
            try:
                monitor.progress(work=1)
            except Cancellation as c:
                # Clean up
                raise c

    return a
```

Note that special caution should be taken to ensure the correct step size, such that the task actually ends when the `total_work` is reached. Apart from progress monitoring it is crucial to implement the possibility to cancel long running operations and perform the appropriate clean up actions when it is cancelled.

Operations that delagate the compute intensive work to `xarray` have often no possibility to report progress in a meaningful way nor to handle cancellation in a timely manner. In this case the `xarray` task can be observed:

```
from cate.core.op import op
from cate.util.monitor import Monitor
import xarray as xr

@op()
def my_op_with_a_monitor(da: xr.DataArray, monitor: Monitor = Monitor.NONE) -> xr.
↪DataArray:
    # Set up the monitor
    with monitor.observing('Monitor Operation'):
      return da.mean(dim='time')
```

See also *Task Monitoring API*.

12.5.6 Adherence to relevant conventions

Cate software often makes the assumption that most if not all of climate data towards which the toolbox is geared adhere to CF Conventions and the Attribute Convention for Data Discovery that both complement each other.

On one hand, an operation may make the assumption that data it receives should be CF compliant. For example, netCDF variables that are ancillary to other variables, such as uncertainty information, should be denoted as such. See CF Ancillary Data.

On the other hand, this implies that special care must be taken to ensure that an operation doesn't break compatibility with said conventions, as well as heeds the advice given in these conventions when creating new variables or datasets.

For example, an operation that adds a mask describing another data variable should follow CF Ancillary Data and CF Flags. Such an operation can be examined in `cate/ops/outliers.py`.

Also, when an operation modifies spatiotemporal extents and/or resolution of the dataset, the corresponding global attributes from Attribute Convention for Data Discovery should be updated or added. There are dedicated functions in `cate/ops/normalize.py` for this purpose.

```
from cate.ops.normalize import adjust_spatial_attrs, adjust_temporal_attrs


@op()
def dummy_op(ds: xr.Dataset):
    rs = ds.copy()

    # Do some science

    # Adjust global attributes
    rs = adjust_spatial_attrs(rs)
    rs = adjust_temporal_attrs(rs)

    return rs
```

12.5.7 Operation outputs

Most operations work on `xr.Datasets` and return these as well. However, some operations may produce information that may be best represented in a tabular form. In these cases it is a good idea to return such data as a `pd.DataFrame` instead of an `xr.Dataset`. This way it can be represented better in the GUI, on the CLI, as well as in Jupyter notebooks.

Cate supports returning multiple named outputs as a Python dictionary.

```
...
@op_output('dataset', data_type=xr.Dataset, description='...')
@op_output('table', data_type=gpd.GeoDataFrame, description='...')
@op_output('scalar', data_type=float, description='...')
def my_op_that_has_named_outputs(...):
  ...
  return {'dataset': ds, 'table': df, 'scalar': x}
```

12.5.8 Using other operations

It is a good idea to use other operations when developing other, more involved operations. Even for seemingly simple cases there might be corner situations that have been solved in the other operation. For example, one is encouraged to use the `subset_spatial` operation as opposed to directly selecting a dataset region using `xr.sel`. Reason being, the given polygon might cross the antimeridian, a situation which is already solved in `cate.ops.subset_spatial`.

Some care must be taken when importing other operations to avoid circular imports. The correct way to import an existing operation is the following:

```
# Directly from subset.py
from cate.ops.subset import subset_spatial
```

12.5.9 Testing

All operations should be well tested. The unit tests should be fast and verify the functionality of the operation, not necessarily validate it. Each module in `cate/ops/` should have the coresponding test module in `test/ops/`. A

bare bones test set up for any operation should be the following:

```
from unittest import TestCase

from cate.core.op import OP_REGISTRY
from cate.util.misc import object_to_qualified_name

from cate.ops import dummy_op


class TestDummyOp(TestCase):
    def test_nominal(self):
        """
        Test nominal execution
        """
        expected = 1
        result = dummy_op()
        self.assertEqual(expected, result)

    def test_error(self):
        """
        Test known error conditions
        """
        with self.assertRaises(ValueError) as err:
            dummy_op(param='will error')
```

It is absolutely crucial to at least have a nominal test that runs the operation with expected inputs that asserts if the outputs is what was expected, the imported operation will automatically be invoked through the operation registry and this will also work in validating if the decorators have been used properly.

If an operation implements a monitor, it is a good idea to test if it has been implemented properly. For example:

```
from unittest import TestCase
from cate.util.monitor import ConsoleMonitor
from cate.ops import dummy_op


class TestDummyOp(TestCase):
    def test_monitor(self):
        m = ConsoleMonitor()
        result = dummy_op(monitor=m)
        self.assertEqual(m._percentage, 100)
```

It is also a good idea to test if the dataset meta information is altered correctly, if newly created data variables have correct attributes, as well as if unexpected inputs are handled correctly.

12.5.10 Optimization

Profiling

If the operation seems to be too slow it should first be profiled to explore the opportunities of potential improvement. The line_profiler package might come in handy here. It can be installed via conda `conda install line_profiler` and then used in a notebook to time individual lines of a given operation as such:

```
import cate.ops as ops
%load_ext line_profiler
%lprun -f ops.some_op result = some_op()
```

A caveat here is that while profiling, the operation being profiled should be undecorated. Otherwise `line_profiler` has trouble finding the source code to test.

Leveraging xarray and dask

When developing operations it should be kept in mind that every operation can potentially work on out-of-memory datasets. Hence one should try to leverage possibilities offered by xarray and dask as much as possible.

For example, an operation producing a statistical quantity of a timeseries for each lon/lat point of a raster could be naively implemented as such:

```
import xarray as xr
from scipy import tricky_stat

def some_op(da: xr.DataArray):
    """
    Run tricky_stat on the given data array
    """
    for i in range(0, len(ds.lon)):
        for j in range(0, len(ds.lat)):
            array = da.isel(lat=j, lon=i).values
            res[i, j] = tricky_stat(array)
```

However, this implementation will yield a heavy performance implication due to the fact that our `xr.DataArray` is likely distributed among many files, parts of which will be read on each `da.isel(lat=j, lon=i).values` invocation resulting in a large overhead in memory and processing time due to io operations.

A better approach would be to use arithmetics and `xarray` ufuncs directly:

```
import xarray as xr

def some_op(da: xr.DataArray):
    """
    Run tricky_stat on the given data array. Influenced by tricky_stat
    scipy implementation.
    """
    da1 = xr.ufuncs.sqrt(da * MAGIC_CONSTANT)
    tricky_stat = da1.mean(dim='time')
    return tricky_stat
```

This second operation has a potential of running several orders of magnitude faster due to minimized amount of io operations, as well as additional optimizations and parallelization occuring behind the scenes in `xarray` and `dask` code.

12.5.11 Documentation

Operation docstrings are used to provide help information in all channels where an operation may be used. It is rendered on the command line when `cate op info some_op` is invoked, it is shown in the appropriate places on the GUI, invoked by users through Python `help()`, as well as published as part of Cate documentation. Hence, it is of utmost importance that the docstring explains well what a particular operation does, as well as documents all input parameters. See also *Docstrings*.

For example:

```
import xarray as xr
import pandas as pd

def doc_op(ds: xr.Dataset, df: pd.DataFrame, magical_const: float):
    """
    This operation carries out a well documented calculation.

    References
    ----------
    'Source <http://www.science.org/documented/calculation>'_

    :param ds: The input dataset used for calculation
    :param df: A dataframe containing auxiliary information
    :param magical_const: Magical constant to use for calculation
    :return: Input dataset with documented calculation applied to it
    """
    # Do some science
    return ds
```

To make sure generated Cate documentation is updated, don't forget to include the operation in `cate/doc/source/api_reference.rst`

If an existing operation name is altered, don't forget to run a search through Cate documentation source to find the possible places where a documentation update is needed.

12.5.12 Operation development checklist

- Is the function registered with the operation registry properly?
- Is the operation set up for import in `cate/ops/__init__.py`?
- Are operation inputs decorated accordingly? E.g., value sets are provided, links between variables and datasets established?
- If one or multiple outputs are `xr.Dataset`, is history information added when appropriate?
- Does the operation use cate typing system so that it can be integrated with the GUI nicely? Both in the function signature and decorators?
- Are inputs validated?
- If the operation can take a while, does it use a monitor and can be cancelled?
- Is the operation a 'well behaved netCDF filter'?
  - If it adds new variables to the netCDF file, do these follow CF conventions?
  - If the operation has the potential of changing spatiotemporal extents and or resolution, does it update the global attributes accordingly?
  - Does the operation drop valuable global or variable attributes when it shouldn't?
- Does the operation produce outputs of appropriate types?
- Are other operations imported correctly if used?
- Is the operation well tested?
  - Is nominal functionality tested?
  - Is the monitor tested?

  - Are the side effects on attributes and other meta information tested?
  - Are error conditions tested?
  - Do the tests run resonably fast?
- Is the operation properly documented?
- Is the operation properly tagged?

When a newly created operation coresponds to this checklist well, it can be said with some certainty that the operation behaves well with respect to the Cate

framework, as well as the wider world.

CHAPTER 13

Terminology

The following table Table 13.1 based on [RD-9] and [RD-10] lists some of the terms used in the CCI Toolbox user interfaces and throughout this documentation.

Table 13.1: CCI Toolbox Terminology

Term	CCI Toolbox Definition
ECV	Umbrella term for geophysical quantity/quantities associated with climate variation and change as well as the impact of climate change onto Earth (e.g. cloud properties).
ECV product	Subdivision of ECVs in long-term data record of values or fields, covering one or more geophysical quantities (e.g. Cloud Water Path).
Geophysical quantity	One physical parameter/variable in that constitutes a time series of observations (e.g. Cloud Liquid Water Path).
Dataset	In-memory representation of data opened from a *data source*. Contains multiple layers of a geophysical quantity or multiple geophysical quantities with multiple layers encompassing e.g. information on temporal and spatial dimensions and localization or uncertainty information.
Data product	Combination of dataset and geophysical quantity incl. uncertainty information (e.g. Cloud Liquid Water Path from L3S Modis merged phase1 v1.0 including uncertainty, standard deviation, number of observations, . . .)
Data store	Offers multiple *data sources*.
Data source	A concrete source for datasets. Provides the *schema* of datasets and other descriptive meta-information about a dataset such as it geo-spatial coverage. Used to open datasets.
Schema	Describes a dataset's structure, contents and data types.

CHAPTER 14

Related Projects

Given here is a limited selection of related projects that have been developed for climate problems or can deal with climate data.

14.1 Iris

Iris seeks to provide a powerful, easy to use, and community-driven Python library for analysing and visualising meteorological and oceanographic data sets.

With Iris you can:

- Use a single API to work on your data, irrespective of its original format.
- Read and write (CF-)netCDF, GRIB, and PP files.
- Easily produce graphs and maps via integration with `matplotlib` and `cartopy`.

14.2 AOCW

The Apache Open Climate Workbench is Python library for common model evaluation tasks as well as a flexible RESTful API, which allows integrating the capabilities of the toolkit into their workflow regardless of language and environment by performing HTTP requests.

14.3 GrADS

The Grid Analysis and Display System, it is a desktop application which reads, manipulates, and visualizes gridded 4D data, provides a command line interface with a proprietary language and interfaces to Python, Perl, IDL, Matlab.

14.4 SNAP

SNAP, the ESA platform for the Sentinel-1, -2, -3 Toolboxes, and SMOS-Box provide numerous specific readers for Earth Observation data and generic data processors (binning, reprojection, collocation, band-maths, etc) which can also be used for ESA CCI Data. A future extension of the CCI Toolbox will allow to use SNAP data readers and processors and therefore allow for interesting synergistic use of CCI data with Level-1 and Level-2 optical, microwave and SAR data.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W