Open Geospatial Consortium

Submission Date 2024-11-21

Approval Date: 2026-03-05

Publication Date: 2026-04-15

External identifier of this OGC® document: <http://www.opengis.net/doc/cp/openeo-processes/1.2>

Internal reference number of this OGC® document: 24-060

Version: 1.2

Category: OGC® Community Practice

Editor: Matthias Mohr

openEO Processes 1.2 Community Practice

Warning

This document is an OGC Member endorsed international Community Practice. This Community Practice was developed outside of the OGC and the originating party may continue to update their work. However, this document is fixed in content. This document is available on a royalty free, non-discriminatory basis.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document source: the original source of this Community Standard is recommended for use: https://processes.openeo.org/1.2.0/

The document below includes links that do not resolve, but which do in the original source.

Document type: OGC® Community Practice
Document subtype:
Document stage: Approved
Document language: English

OGC openEO Processes Community Standard (v1.2.0)

Search in processes

Show ☐ deprecated ☐ experimental

- **Aggregate & Resample (5/8)**
- **Arrays (12/19)**
- **Climatology (3)**
- **Comparison (15/16)**
- **Cubes (27/41)**
- **Export (1)**
- **Filter (5/6)**
- **Import (2/5)**
- **Logic (7)**
- **Masks (3)**
- **Math (17/20)**
- **Math > Constants (4/5)**
- **Math > Exponential & Logarithmic (6)**
- **Math > Image Filter (1)**
- **Math > Indices (2)**
- **Math > Rounding (4)**
- **Math > Statistics (9)**
- **Math > Trigonometric (14)**
- **Reducer (17/19)**
- **Sorting (3)**
- **Texts (6)**
- **Udf (1/2)**
- **Vegetation Indices (2)**

Introduction

The OGC openEO Community Standard defines a set of well-defined processes in support of interoperable cloud-based processing of large Earth observation datasets.

We recommend reading the glossary before diving into this specification. The glossary explains the most important terms used in this specification.

The OGC openEO Community Standard consists of two parts:

- OGC openEO API Community Standard, the accompanying API specification that enables discovery, chaining and execution of the processes defined in this sepcification
- OGC openEO Processes Community Standard, this specification

Abstract

openEO specifies an open application programming interface (API) for connecting applications and other client software to big Earth observation cloud back-ends in a simple and unified way.

The openEO specification aims at increasing the interoperability of big EO data processing of satellite imagery in the cloud. Implementations of openEO can be used to add an interoperability layer on top of existing services. Its development has been driven by the need to overcome the challenges associated with different tools, APIs, and data formats in geospatial technology. openEO has been developed from the bottom up, with each version of the specification supported by implementations.

The primary use case for specifying openEO was to simplify and unify the data processing using a common API and a specification for a set of pre-defined processes. As such, users can still work in their favored programming language without worrying about data organization and pre-processing. Users can avoid vendor lock-in as the generated process descriptions can be executed at multiple provider endpoints, making it easier to compare and reproduce processing results between different providers.

Source of this Document

The majority of the content in this OGC document is a direct copy of the content contained at https://github.com/Open-EO/openeo-processes. No normative changes have been made to the content. This OGC document does contain content not in source openEO Processes GitHub repository. Specifically, while derived from content on the openEO Processes repository, the chapters "Abstract", "Source of this Document", "Submitting Organizations", and "Supporting Organizations" in this document are not found on the openEO Processes repository.

Submitting Organizations

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- openEO Project Steering Committee

The organizations listed above have granted the Open Geospatial Consortium (OGC) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version under a Apache License, Version 2.0 (see below).

Supporting Organizations

The following organization (in alphabetical order) support the submission of the openEO Community Standard to the OGC:

- EOX IT Services GmbH
- EUMETSAT
- Eurac Research
- European Space Agency (ESA)
- GeoConnections - Natural Resources Canada
- German Aerospace Center - DLR
- Matthias Mohr - Softwareentwicklung
- Planet Labs PBC
- Telespazio VEGA UK Ltd
- University of Münster - Institute for Geoinformatics
- VITO (Flemish Institute for Technological Research)

License Agreement

absolute

Absolute value

Download JSON

Description

`absolute(number|null x) : number|null`

Computes the absolute value of a real number `x`, which is the "unsigned" portion of x and often denoted as *|x|*.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed absolute value.

> Data type: **number, null**
>
> Minimum value (inclusive): 0

Examples

Example #1

```
absolute(x = 0) => 0
```

Example #2

```
absolute(x = 3.5) => 3.5
```

Example #3

```
absolute(x = -0.4) => 0.4
```

Example #4

```
absolute(x = -3.5) => 3.5
```

See Also

- Absolute value explained by Wolfram MathWorld

add

Addition of two numbers

MATH

Download JSON

Description

`add(number|null x, number|null y) : number|null`

Sums up the two numbers `x` and `y` (*`x + y`*) and returns the computed sum.

No-data values are taken into account so that `null` is returned if any element is such a value.

The computations follow IEEE Standard 754 whenever the processing environment supports it.

Parameters

x*

The first summand.

> Data type: **number, null**

y*

The second summand.

> Data type: **number, null**

Return Value

The computed sum of the two numbers.

> Data type: **number, null**

Examples

Example #1

```
add(x = 5, y = 2.5) => 7.5
```

Example #2

```
add(x = -2, y = -4) => -6
```

Example #3

```
add(x = 1, y = null) => null
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Sum explained by Wolfram MathWorld

add_dimension

Add a new dimension

CUBES

Download JSON

Description

```
add_dimension(raster-cube data, string name, number|string label, ?string type = "other") : raster-cube
```

Adds a new named dimension to the data cube.

Afterwards, the dimension can be referred to with the specified `name`. If a dimension with the specified name exists, the process fails with a `DimensionExists` exception. The dimension label of the dimension is set to the specified `label`.

Parameters

data*

A data cube to add the dimension to.

> Data type: **raster-cube**

name*

Name for the dimension.

> Data type: **string**

label*

A dimension label.

Data Types:

> Data type: **number**

> Data type: **string**

type = "other"

The type of dimension, defaults to `other`.

> Data type: **string**
>
> Allowed values: spatial, temporal, bands, other

Return Value

The data cube with a newly added dimension. The new dimension has exactly one dimension label. All other dimensions remain unchanged.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionExists`
  Message: *A dimension with the specified name already exists.*

aggregate_spatial

Zonal statistics for geometries

CUBES AGGREGATE & RESAMPLE

Download JSON

Description

```
aggregate_spatial(raster-cube data, geojson:object geometries, process-graph:object reducer,
?string target_dimension = "result", ?any context = null) : vector-cube
```

Aggregates statistics for one or more geometries (e.g. zonal statistics for polygons) over the spatial dimensions. The number of total and valid pixels is returned together with the calculated values.

An 'unbounded' aggregation over the full extent of the horizontal spatial dimensions can be computed with the process `reduce_spatial`.

This process passes a list of values to the reducer. The list of values has an undefined order, therefore processes such as `last` and `first` that depend on the order of the values will lead to unpredictable results.

Parameters

`data`*

A raster data cube.

The data cube must have been reduced to only contain two spatial dimensions and a third dimension the values are aggregated for, for example the temporal dimension to get a time series. Otherwise, this process fails with the `TooManyDimensions` exception.

The data cube implicitly gets restricted to the bounds of the geometries as if `filter_spatial` would have been used with the same values for the corresponding parameters immediately before this process.

> Data type: **raster-cube**

geometries*

Geometries as GeoJSON on which the aggregation will be based.

One value will be computed per GeoJSON `Feature`, `Geometry` or `GeometryCollection`. For a `FeatureCollection` multiple values will be computed, one value per contained `Feature`. For example, a single value will be computed for a `MultiPolygon`, but two values will be computed for a `FeatureCollection` containing two polygons.

- For **polygons**, the process considers all pixels for which the point at the pixel center intersects with the corresponding polygon (as defined in the Simple Features standard by the OGC).
- For **points**, the process considers the closest pixel center.
- For **lines** (line strings), the process considers all the pixels whose centers are closest to at least one point on the line.

Thus, pixels may be part of multiple geometries and be part of multiple aggregations.

To maximize interoperability, a nested `GeometryCollection` should be avoided. Furthermore, a `GeometryCollection` composed of a single type of geometries should be avoided in favour of the corresponding multi-part type (e.g. `MultiPolygon`).

> Data type: **geojson:object**

reducer*

A reducer to be applied on all values of each geometry. A reducer is a single process such as `mean` or a set of processes, which computes a single value for a list of values, see the category 'reducer' for such processes.

> Data type: **User-defined Process (process-graph:object)**
>
> **Parameters:**
>
> ### data*
>
> An array with elements of any type.
>
> > Data type: **array**
> >
> > Array items:
> >
> > > Any data type.
> > >
> > > Data type: **any**
>
> ### context = null
>
> Additional data passed by the user.
>
> > Any data type.
> >
> > Data type: **any**
>
> **Expected Return Value:**
>
> The value to be set in the vector data cube.

> Any data type.
>
> Data type: **any**

`target_dimension = "result"`

The new dimension name to be used for storing the results. Defaults to `result`.

> Data type: **string**

`context = null`

Additional data to be passed to the reducer.

> Any data type.
>
> Data type: **any**

Return Value

A vector data cube with the computed results and restricted to the bounds of the geometries.

The computed value is used for the dimension with the name that was specified in the parameter `target_dimension`.

The computation also stores information about the total count of pixels (valid + invalid pixels) and the number of valid pixels (see `is_valid`) for each geometry. These values are added as a new dimension with a dimension name derived from `target_dimension` by adding the suffix `_meta`. The new dimension has the dimension labels `total_count` and `valid_count`.

> Data type: **vector-cube**

Errors/Exceptions

- `TooManyDimensions`
  Message: *The number of dimensions must be reduced to three for `aggregate_spatial`.*

See Also

- Aggregation explained in the openEO documentation
- Simple Features standard by the OGC

aggregate_spatial_window

Zonal statistics for rectangular windows — **experimental**

CUBES AGGREGATE & RESAMPLE

Download JSON

Description

```
aggregate_spatial_window(raster-cube data, process-graph:object reducer, array<integer> size,
?string boundary = "pad", ?string align = "upper-left", ?any context = null) : raster-cube
```

Aggregates statistics over the horizontal spatial dimensions (axes `x` and `y`) of the data cube.

The pixel grid for the axes `x` and `y` is divided into non-overlapping windows with the size specified in the parameter `size`. If the number of values for the axes `x` and `y` is not a multiple of the corresponding window size, the behavior specified in the parameters `boundary` and `align` is applied. For each of these windows, the reducer process computes the result.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

A raster data cube with exactly two horizontal spatial dimensions and an arbitrary number of additional dimensions. The process is applied to all additional dimensions individually.

Data type: **raster-cube**

`reducer`*

A reducer to be applied on the list of values, which contain all pixels covered by the window. A reducer is a single process such as `mean` or a set of processes, which computes a single value for a list of values, see the category 'reducer' for such processes.

Data type: **User-defined Process (process-graph:object)**

Parameters:

`data`*

An array with elements of any type.

Data type: **array**

Array items:

Any data type.

Data type: **any**

`context` = `null`

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new data cube.

Any data type.

Data type: **any**

`size`*

Window size in pixels along the horizontal spatial dimensions.

The first value corresponds to the `x` axis, the second value corresponds to the `y` axis.

Data type: **array<integer>**

Min. number of items: 2

Max. number of items: 2

Array items:

Data type: **integer**

Minimum value (inclusive): 1

`boundary` = `"pad"`

Behavior to apply if the number of values for the axes `x` and `y` is not a multiple of the corresponding value in the `size` parameter. Options are:

- `pad` (default): pad the data cube with the no-data value `null` to fit the required window size.
- `trim`: trim the data cube to fit the required window size.

Set the parameter `align` to specifies to which corner the data is aligned to.

Data type: **string**

Allowed values: pad, trim

`align` = `"upper-left"`

If the data requires padding or trimming (see parameter `boundary`), specifies to which corner of the spatial extent the data is aligned to. For example, if the data is aligned to the upper left, the process pads/trims at the lower-right.

Data type: **string**

Allowed values: lower-left, upper-left, lower-right, upper-right

`context` = `null`

Additional data to be passed to the reducer.

Any data type.

Data type: **any**

Return Value

A data cube with the newly computed values and the same dimensions.

The resolution will change depending on the chosen values for the `size` and `boundary` parameter. It usually decreases for the dimensions which have the corresponding parameter `size` set to values greater than 1.

The dimension labels will be set to the coordinate at the center of the window. The other dimension properties (name, type and reference system) remain unchanged.

> Data type: **raster-cube**

See Also

- Aggregation explained in the openEO documentation

aggregate_temporal

Temporal aggregations

CUBES AGGREGATE & RESAMPLE

Download JSON

Description

```
aggregate_temporal(raster-cube data, temporal-intervals:array<temporal-
interval:array<string|null>> intervals, process-graph:object reducer, ?array<number|string>
labels = [], ?string|null dimension = null, ?any context = null) : raster-cube
```

Computes a temporal aggregation based on an array of temporal intervals.

For common regular calendar hierarchies such as year, month, week or seasons `aggregate_temporal_period` can be used. Other calendar hierarchies must be transformed into specific intervals by the clients.

For each interval, all data along the dimension will be passed through the reducer.

The computed values will be projected to the labels. If no labels are specified, the start of the temporal interval will be used as label for the corresponding values. In case of a conflict (i.e. the user-specified values for the start times of the temporal intervals are not distinct), the user-defined labels must be specified in the parameter `labels` as otherwise a `DistinctDimensionLabelsRequired` exception would be thrown. The number of user-defined labels and the number of intervals need to be equal.

If the dimension is not set or is set to `null`, the data cube is expected to only have one temporal dimension.

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`intervals`*

Left-closed temporal intervals, which are allowed to overlap. Each temporal interval in the array has exactly two elements:

1. The first element is the start of the temporal interval. The specified instance in time is **included** in the interval.
2. The second element is the end of the temporal interval. The specified instance in time is **excluded** from the interval.

The specified temporal strings follow RFC 3339. Although RFC 3339 prohibits the hour to be '24', **this process allows the value '24' for the hour** of an end time in order to make it possible that left-closed time intervals can fully cover the day.

Data type: **temporal-intervals:array<temporal-interval:array<date-time:string|date:string|time:string|year:string|null>>**

Min. number of items: 1

Array items:

Data type: **temporal-interval:array<date-time:string|date:string|time:string|year:string|null>**

Min. number of items: 2

Max. number of items: 2

Array items: Data type: **any**

Examples:

- `[ [ "2015-01-01", "2016-01-01" ], [ "2016-01-01", "2017-01-01" ], [ "2017-01-01", "2018-01-01" ] ]`
- `[ [ "00:00:00Z", "12:00:00Z" ], [ "12:00:00Z", "24:00:00Z" ] ]`

reducer*

A reducer to be applied for the values contained in each interval. A reducer is a single process such as `mean` or a set of processes, which computes a single value for a list of values, see the category 'reducer' for such processes. Intervals may not contain any values, which for most reducers leads to no-data (`null`) values by default.

Data type: **User-defined Process (process-graph:object)**

Parameters:

data*

A labeled array with elements of any type. If there's no data for the interval, the array is empty.

Data type: **labeled-array**

Array items:

Any data type.

Data type: **any**

context = null

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new data cube.

Any data type.

Data type: **any**

`labels = []`

Distinct labels for the intervals, which can contain dates and/or times. Is only required to be specified if the values for the start of the temporal intervals are not distinct and thus the default labels would not be unique. The number of labels and the number of groups need to be equal.

Data type: **array<number|string>**

`dimension = null`

The name of the temporal dimension for aggregation. All data along the dimension is passed through the specified reducer. If the dimension is not set or set to `null`, the data cube is expected to only have one temporal dimension. Fails with a `TooManyDimensions` exception if it has more dimensions. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

Data type: **string, null**

`context = null`

Additional data to be passed to the reducer.

Any data type.

Data type: **any**

Return Value

A new data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged, except for the resolution and dimension labels of the given temporal dimension.

Data type: **raster-cube**

Errors/Exceptions

- `TooManyDimensions`
  Message: *The data cube contains multiple temporal dimensions. The parameter `dimension` must be specified.*
- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*
- `DistinctDimensionLabelsRequired`
  Message: *The dimension labels have duplicate values. Distinct labels must be specified.*

Examples

Example #1

```
aggregate_temporal(data = $data, intervals = [["2015-01-01","2016-01-01"],["2016-01-01","2017-01-01"],["2017-01-01","2018-01-01"],["2018-01-01","2019-01-01"],["2019-01-01","2020-01-01"]], reducer = {"process_graph":{"mean1":{"process_id":"mean","arguments":{"data":{"from_parameter":"data"}},"result":true}}}, labels = ["2015","2016","2017","2018","2019"])
```

See Also

- Aggregation explained in the openEO documentation

aggregate_temporal_period

Temporal aggregations based on calendar hierarchies

AGGREGATE & RESAMPLE CLIMATOLOGY CUBES

Download JSON

Description

```
aggregate_temporal_period(raster-cube data, string period, process-graph:object reducer, ?string|null dimension = null, ?any context = null) : raster-cube
```

Computes a temporal aggregation based on calendar hierarchies such as years, months or seasons. For other calendar hierarchies `aggregate_temporal` can be used.

For each interval, all data along the dimension will be passed through the reducer.

If the dimension is not set or is set to `null`, the data cube is expected to only have one temporal dimension.

Parameters

`data`*

The source data cube.

Data type: **raster-cube**

`period`*

The time intervals to aggregate. The following pre-defined values are available:

- `hour`: Hour of the day
- `day`: Day of the year
- `week`: Week of the year
- `dekad`: Ten day periods, counted per year with three periods per month (day 1 - 10, 11 - 20 and 21 - end of month). The third dekad of the month can range from 8 to 11 days. For example, the fourth dekad is Feb, 1 - Feb, 10 each year.

- `month`: Month of the year
- `season`: Three month periods of the calendar seasons (December - February, March - May, June - August, September - November).
- `tropical-season`: Six month periods of the tropical seasons (November - April, May - October).
- `year`: Proleptic years
- `decade`: Ten year periods (0-to-9 decade), from a year ending in a 0 to the next year ending in a 9.
- `decade-ad`: Ten year periods (1-to-0 decade) better aligned with the anno Domini (AD) calendar era, from a year ending in a 1 to the next year ending in a 0.

Data type: **string**

Allowed values: hour, day, week, dekad, month, season, tropical-season, year, decade, decade-ad

reducer*

A reducer to be applied for the values contained in each period. A reducer is a single process such as `mean` or a set of processes, which computes a single value for a list of values, see the category 'reducer' for such processes. Periods may not contain any values, which for most reducers leads to no-data (`null`) values by default.

Data type: **User-defined Process (process-graph:object)**

Parameters:

data*

A labeled array with elements of any type. If there's no data for the period, the array is empty.

Data type: **labeled-array**

Array items:

Any data type.

Data type: **any**

context = null

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new data cube.

Any data type.

Data type: **any**

dimension = null

The name of the temporal dimension for aggregation. All data along the dimension is passed through the specified reducer. If the dimension is not set or set to `null`, the source data cube is expected to only have one temporal dimension. Fails with a `TooManyDimensions` exception if it has more dimensions. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

> Data type: **string, null**

`context` = `null`

Additional data to be passed to the reducer.

> Any data type.
>
> Data type: **any**

Return Value

A new data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged, except for the resolution and dimension labels of the given temporal dimension. The specified temporal dimension has the following dimension labels (`YYYY` = four-digit year, `MM` = two-digit month, `DD` two-digit day of month):

- `hour`: `YYYY-MM-DD-00` - `YYYY-MM-DD-23`
- `day`: `YYYY-001` - `YYYY-365`
- `week`: `YYYY-01` - `YYYY-52`
- `dekad`: `YYYY-00` - `YYYY-36`
- `month`: `YYYY-01` - `YYYY-12`
- `season`: `YYYY-djf` (December - February), `YYYY-mam` (March - May), `YYYY-jja` (June - August), `YYYY-son` (September - November).
- `tropical-season`: `YYYY-ndjfma` (November - April), `YYYY-mjjaso` (May - October).
- `year`: `YYYY`
- `decade`: `YYY0`
- `decade-ad`: `YYY1`

The dimension labels in the new data cube are complete for the whole extent of the source data cube. For example, if `period` is set to `day` and the source data cube has two dimension labels at the beginning of the year (`2020-01-01`) and the end of a year (`2020-12-31`), the process returns a data cube with 365 dimension labels (`2020-001`, `2020-002`, ..., `2020-365`). In contrast, if `period` is set to `day` and the source data cube has just one dimension label `2020-01-05`, the process returns a data cube with just a single dimension label (`2020-005`).

> Data type: **raster-cube**

Errors/Exceptions

- `TooManyDimensions`

  Message: *The data cube contains multiple temporal dimensions. The parameter `dimension` must be specified.*
- `DimensionNotAvailable`

  Message: *A dimension with the specified name does not exist.*
- `DistinctDimensionLabelsRequired`

  Message: *The dimension labels have duplicate values. Distinct labels must be specified.*

See Also

- Aggregation explained in the openEO documentation

all

Are all of the values true?

LOGIC REDUCER

Download JSON

Description

```
all(array<boolean|null> data, ?boolean ignore_nodata = true) : boolean|null
```

Checks if **all** of the values in `data` are true. If no value is given (i.e. the array is empty) the process returns `null`.

By default all no-data values are ignored so that the process returns `null` if all values are no-data, `true` if all values are true and `false` otherwise. Setting the `ignore_nodata` flag to `false` takes no-data values into account and the array values are reduced pairwise according to the following truth table:

```
      || null  | false | true
----- || ----- | ----- | -----
null  || null  | false | null
false || false | false | false
true  || null  | false | true
```

Remark: The process evaluates all values from the first to the last element and stops once the outcome is unambiguous. A result is ambiguous unless a value is `false` or all values have been taken into account.

Parameters

data*

A set of boolean values.

> Data type: **array<boolean|null>**

ignore_nodata = true

Indicates whether no-data values are ignored or not and ignores them by default.

> Data type: **boolean**

Return Value

Boolean result of the logical operation.

> Data type: **boolean, null**

Examples

Example #1

```
all(data = [false,null]) => false
```

Example #2

```
all(data = [true,null]) => true
```

Example #3

```
all(data = [false,null], ignore_nodata = false) => false
```

Example #4

```
all(data = [true,null], ignore_nodata = false) => null
```

Example #5

```
all(data = [true,false,true,false]) => false
```

Example #6

```
all(data = [true,false]) => false
```

Example #7

```
all(data = [true,true]) => true
```

Example #8

```
all(data = [true]) => true
```

Example #9

```
all(data = [null], ignore_nodata = false) => null
```

Example #10

```
all(data = []) => null
```

and

Logical AND

LOGIC

Download JSON

Description

```
and(boolean|null x, boolean|null y) : boolean|null
```

Checks if **both** values are true.

Evaluates parameter `x` before `y` and stops once the outcome is unambiguous. If any argument is `null`, the result will be `null` if the outcome is ambiguous.

Truth table:

```
a \ b || null  | false | true
----- || ----- | ----- | -----
null  || null  | false | null
false || false | false | false
true  || null  | false | true
```

Parameters

x*

A boolean value.

> Data type: **boolean, null**

y*

A boolean value.

> Data type: **boolean, null**

Return Value

Boolean result of the logical AND.

> Data type: **boolean, null**

Examples

Example #1

```
and(x = true, y = true) => true
```

Example #2

```
and(x = true, y = false) => false
```

Example #3

```
and(x = false, y = false) => false
```

Example #4

```
and(x = false, y = null) => false
```

Example #5

```
and(x = true, y = null) => null
```

anomaly

Compute anomalies

CLIMATOLOGY MATH

Download JSON

Description

```
anomaly(raster-cube data, raster-cube normals, string period) : raster-cube
```

Computes anomalies based on normals for temporal periods. It compares the data for each label in the temporal dimension with the corresponding data in the normals data cube by subtracting the normal from the data.

Parameters

data*

A data cube with exactly one temporal dimension and the following dimension labels for the given period (`YYYY` = four-digit year, `MM` = two-digit month, `DD` two-digit day of month):

- `hour`: `YYYY-MM-DD-00` - `YYYY-MM-DD-23`
- `day`: `YYYY-001` - `YYYY-365`
- `week`: `YYYY-01` - `YYYY-52`
- `dekad`: `YYYY-00` - `YYYY-36`
- `month`: `YYYY-01` - `YYYY-12`
- `season`: `YYYY-djf` (December - February), `YYYY-mam` (March - May), `YYYY-jja` (June - August), `YYYY-son` (September - November).
- `tropical-season`: `YYYY-ndjfma` (November - April), `YYYY-mjjaso` (May - October).
- `year`: `YYYY`
- `decade`: `YYY0`
- `decade-ad`: `YYY1`
- `single-period` / `climatology-period`: Any

`aggregate_temporal_period` can compute such a data cube.

> Data type: **raster-cube**

normals*

A data cube with normals, e.g. daily, monthly or yearly values computed from a process such as `climatological_normal`. Must contain exactly one temporal dimension with the following dimension labels for the given period:

- `hour`: `00` - `23`
- `day`: `001` - `365`
- `week`: `01` - `52`
- `dekad`: `00` - `36`
- `month`: `01` - `12`
- `season`: `djf` (December - February), `mam` (March - May), `jja` (June - August), `son` (September - November)
- `tropical-season`: `ndjfma` (November - April), `mjjaso` (May - October)
- `year`: Four-digit year numbers
- `decade`: Four-digit year numbers, the last digit being a `0`
- `decade-ad`: Four-digit year numbers, the last digit being a `1`
- `single-period` / `climatology-period`: A single dimension label with any name is expected.

> Data type: **raster-cube**

`period`*

Specifies the time intervals available in the normals data cube. The following options are available:

- `hour`: Hour of the day
- `day`: Day of the year
- `week`: Week of the year
- `dekad`: Ten day periods, counted per year with three periods per month (day 1 - 10, 11 - 20 and 21 - end of month). The third dekad of the month can range from 8 to 11 days. For example, the fourth dekad is Feb, 1 - Feb, 10 each year.
- `month`: Month of the year
- `season`: Three month periods of the calendar seasons (December - February, March - May, June - August, September - November).
- `tropical-season`: Six month periods of the tropical seasons (November - April, May - October).
- `year`: Proleptic years
- `decade`: Ten year periods (0-to-9 decade), from a year ending in a 0 to the next year ending in a 9.
- `decade-ad`: Ten year periods (1-to-0 decade) better aligned with the anno Domini (AD) calendar era, from a year ending in a 1 to the next year ending in a 0.
- `single-period` / `climatology-period`: A single period of arbitrary length

> Data type: **string**
>
> Allowed values: hour, day, week, dekad, month, season, tropical-season, year, decade, decade-ad, climatology-period, single-period

Return Value

A data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

> Data type: **raster-cube**

any

Is at least one value true?

LOGIC REDUCER

Download JSON

Description

```
any(array<boolean|null> data, ?boolean ignore_nodata = true) : boolean|null
```

Checks if **any** (i.e. at least one) value in `data` is `true`. If no value is given (i.e. the array is empty) the process returns `null`.

By default all no-data values are ignored so that the process returns `null` if all values are no-data, `true` if at least one value is true and `false` otherwise. Setting the `ignore_nodata` flag to `false` takes no-data values into account and the array values are reduced pairwise according to the following truth table:

```
      || null | false | true
----- || ---- | ----- | ----
null  || null | null  | true
false || null | false | true
true  || true | true  | true
```

Remark: The process evaluates all values from the first to the last element and stops once the outcome is unambiguous. A result is ambiguous unless a value is `true`.

Parameters

`data`*

A set of boolean values.

> Data type: **array<boolean|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not and ignores them by default.

> Data type: **boolean**

Return Value

Boolean result of the logical operation.

> Data type: **boolean, null**

Examples

Example #1

```
any(data = [false,null]) => false
```

Example #2

```
any(data = [true,null]) => true
```

Example #3

```
any(data = [false,null], ignore_nodata = false) => null
```

Example #4

```
any(data = [true,null], ignore_nodata = false) => true
```

Example #5

```
any(data = [true,false,true,false]) => true
```

Example #6

```
any(data = [true,false]) => true
```

Example #7

```
any(data = [false,false]) => false
```

Example #8

```
any(data = [true]) => true
```

Example #9

```
any(data = [null], ignore_nodata = false) => null
```

Example #10

```
any(data = []) => null
```

apply

Apply a process to each pixel

CUBES

Download JSON

Description

```
apply(raster-cube data, process-graph:object process, ?any context = null) : raster-cube
```

Applies a process to each pixel value in the data cube (i.e. a local operation). In contrast, the process `apply_dimension` applies a process to all pixel values along a particular dimension.

Parameters

data*

A data cube.

> Data type: **raster-cube**

process*

A process that accepts and returns a single value and is applied on each individual value in the data cube. The process may consist of multiple sub-processes and could, for example, consist of processes such as `abs` or `linear_scale_range`.

Data type: **User-defined Process (process-graph:object)**

Parameters:

x*

The value to process.

Any data type.

Data type: **any**

`context = null`

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new data cube.

Any data type.

Data type: **any**

`context = null`

Additional data to be passed to the process.

Any data type.

Data type: **any**

Return Value

A data cube with the newly computed values and the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

Data type: **raster-cube**

See Also

- Apply explained in the openEO documentation

apply_dimension

Apply a process to pixels along a dimension

CUBES

Download JSON

Description

```
apply_dimension(raster-cube data, process-graph:object process, string dimension, ?string|null target_dimension = null, ?any context = null) : raster-cube
```

Applies a process to all pixel values along a dimension of a raster data cube. For example, if the temporal dimension is specified the process will work on a time series of pixel values.

The process `reduce_dimension` also applies a process to pixel values along a dimension, but drops the dimension afterwards. The process `apply` applies a process to each pixel value in the data cube.

The target dimension is the source dimension if not specified otherwise in the `target_dimension` parameter. The pixel values in the target dimension get replaced by the computed pixel values. The name, type and reference system are preserved.

The dimension labels are preserved when the target dimension is the source dimension and the number of pixel values in the source dimension is equal to the number of values computed by the process. Otherwise, the dimension labels will be incrementing integers starting from zero, which can be changed using `rename_labels` afterwards. The number of labels will equal to the number of values computed by the process.

Parameters

data*

A data cube.

Data type: **raster-cube**

process*

Process to be applied on all pixel values. The specified process needs to accept an array and must return an array with at least one element. A process may consist of multiple sub-processes.

Data type: **User-defined Process (process-graph:object)**

Parameters:

data*

A labeled array with elements of any type.

Data type: **labeled-array**

Array items:

Any data type.

Data type: **any**

context = null

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:
The value to be set in the new data cube.

Data type: **array**

Array items:

Any data type.

Data type: **any**

`dimension`*

The name of the source dimension to apply the process on. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

Data type: **string**

`target_dimension = null`

The name of the target dimension or `null` (the default) to use the source dimension specified in the parameter `dimension`.

By specifying a target dimension, the source dimension is removed. The target dimension with the specified name and the type `other` (see `add_dimension`) is created, if it doesn't exist yet.

Data type: **string, null**

`context = null`

Additional data to be passed to the process.

Any data type.

Data type: **any**

Return Value

A data cube with the newly computed values.

All dimensions stay the same, except for the dimensions specified in corresponding parameters. There are three cases how the dimensions can change:

1. The source dimension is the target dimension:
   - The (number of) dimensions remain unchanged as the source dimension is the target dimension.
   - The source dimension properties name and type remain unchanged.

    - The dimension labels, the reference system and the resolution are preserved only if the number of pixel values in the source dimension is equal to the number of values computed by the process. Otherwise, all other dimension properties change as defined in the list below.
2. The source dimension is not the target dimension and the latter exists:
    - The number of dimensions decreases by one as the source dimension is dropped.
    - The target dimension properties name and type remain unchanged. All other dimension properties change as defined in the list below.
3. The source dimension is not the target dimension and the latter does not exist:
    - The number of dimensions remain unchanged, but the source dimension is replaced with the target dimension.
    - The target dimension has the specified name and the type other. All other dimension properties are set as defined in the list below.

Unless otherwise stated above, for the given (target) dimension the following applies:

- the number of dimension labels is equal to the number of values computed by the process,
- the dimension labels are incrementing integers starting from zero,
- the resolution changes, and
- the reference system is undefined.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

See Also

- Apply explained in the openEO documentation

apply_kernel

Apply a spatial convolution with a kernel

CUBES MATH > IMAGE FILTER Download JSON

Description

```
apply_kernel(raster-cube data, kernel:array<array<number>> kernel, ?number factor = 1, ?
string|number border = 0, ?number replace_invalid = 0) : raster-cube
```

Applies a 2D convolution (i.e. a focal operation with a weighted kernel) on the horizontal spatial dimensions (axes `x` and `y`) of the data cube.

Each value in the kernel is multiplied with the corresponding pixel value and all products are summed up afterwards. The sum is then multiplied with the factor.

The process can't handle non-numerical or infinite numerical values in the data cube. Boolean values are converted to integers (`false` = 0, `true` = 1), but all other non-numerical or infinite values are replaced with zeroes by default (see parameter `replace_invalid`).

For cases requiring more generic focal operations or non-numerical values, see `apply_neighborhood`.

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`kernel`*

Kernel as a two-dimensional array of weights. The inner level of the nested array aligns with the `x` axis and the outer level aligns with the `y` axis. Each level of the kernel must have an uneven number of elements, otherwise the process throws a `KernelDimensionsUneven` exception.

> A two-dimensional array of numbers.
>
> Data type: **kernel:array<array<number>>**
>
> Array items:
>
> > Data type: **array<number>**
> >
> > Array items:
> >
> > > Data type: **number**

`factor` = 1

A factor that is multiplied to each value after the kernel has been applied.

This is basically a shortcut for explicitly multiplying each value by a factor afterwards, which is often required for some kernel-based algorithms such as the Gaussian blur.

> Data type: **number**

`border` = 0

Determines how the data is extended when the kernel overlaps with the borders. Defaults to fill the border with zeroes.

The following options are available:

- *numeric value* - fill with a user-defined constant number `n`: `nnnnnn|abcdefgh|nnnnnn` (default, with `n` = 0)
- `replicate` - repeat the value from the pixel at the border: `aaaaaa|abcdefgh|hhhhhh`
- `reflect` - mirror/reflect from the border: `fedcba|abcdefgh|hgfedc`
- `reflect_pixel` - mirror/reflect from the center of the pixel at the border: `gfedcb|abcdefgh|gfedcb`
- `wrap` - repeat/wrap the image: `cdefgh|abcdefgh|abcdef`

Data Types:

> Data type: **string**
>
> Allowed values: replicate, reflect, reflect_pixel, wrap

> Data type: **number**

`replace_invalid = 0`

This parameter specifies the value to replace non-numerical or infinite numerical values with. By default, those values are replaced with zeroes.

> Data type: **number**

Return Value

A data cube with the newly computed values and the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

> Data type: **raster-cube**

Errors/Exceptions

- `KernelDimensionsUneven`
  Message: *Each dimension of the kernel must have an uneven number of elements.*

See Also

- Apply explained in the openEO documentation
- Convolutions explained
- Example of 2D Convolution

apply_neighborhood

Apply a process to pixels in a n-dimensional neighborhood

CUBES

Download JSON

Description

```
apply_neighborhood(raster-cube data, process-graph:object process, array<chunk-size:object>
size, ?array<chunk-size:object> overlap, ?any context = null) : raster-cube
```

Applies a focal process to a data cube.

A focal process is a process that works on a 'neighborhood' of pixels. The neighborhood can extend into multiple dimensions, this extent is specified by the `size` argument. It is not only (part of) the size of the input window, but also the size of the output for a given position of the sliding window. The sliding window moves with multiples of `size`.

An overlap can be specified so that neighborhoods can have overlapping boundaries. This allows for continuity of the output. The values included in the data cube as overlap can't be modified by the given `process`. The missing overlap at the borders of the original data cube are made available as no-data (`null`) in the sub data cubes.

The neighborhood size should be kept small enough, to avoid running beyond computational resources, but a too small size will result in a larger number of process invocations, which may slow down processing. Window sizes for spatial dimensions typically are in the range of 64 to 512 pixels, while overlaps of 8 to 32 pixels are common.

The process must not add new dimensions, or remove entire dimensions, but the result can have different dimension labels.

For the special case of 2D convolution, it is recommended to use `apply_kernel`.

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`process`*

Process to be applied on all neighborhoods.

> Data type: **User-defined Process (process-graph:object)**
>
> **Parameters:**
>
> #### `data`*
>
> A subset of the data cube as specified in `context` and `overlap`.
>
> > Data type: **raster-cube**
>
> #### `context` = `null`
>
> Additional data passed by the user.
>
> > Any data type.
> >
> > Data type: **any**
>
> **Expected Return Value:**
>
> The data cube with the newly computed values and the same dimensions. The dimension properties (name, type, labels, reference system and resolution) must remain unchanged, otherwise a `DataCubePropertiesImmutable` exception will be thrown.
>
> > Data type: **raster-cube**

`size`*

Neighborhood sizes along each dimension.

This object maps dimension names to either a physical measure (e.g. 100 m, 10 days) or pixels (e.g. 32 pixels). For dimensions not specified, the default is to provide all values. Be aware that including all values from overly large dimensions may not be processed at once.

> Data type: **array<chunk-size:object>**
>
> Array items:
>
> > Data type: **chunk-size:object**
> >
> > **Object Properties:**
> >
> > ***dimension*** * Data type: **string**

***value** **

Data type: **any**

Default value: `null`

unit

The unit the values are given in, either in meters (`m`) or pixels (`px`). If no unit is given, uses the unit specified for the dimension or otherwise the default unit of the reference system.

Data type: **string**

Allowed values: px, m

`overlap`

Overlap of neighborhoods along each dimension to avoid border effects.

For instance a temporal dimension can add 1 month before and after a neighborhood. In the spatial dimensions, this is often a number of pixels. The overlap specified is added before and after, so an overlap of 8 pixels will add 8 pixels on both sides of the window, so 16 in total.

Be aware that large overlaps increase the need for computational resources and modifying overlapping data in subsequent operations have no effect.

Data type: **array<chunk-size:object>**

Array items:

Data type: **chunk-size:object**

Object Properties:

***dimension** **

Data type: **string**

***value** **

Data type: **any**

Default value: `null`

unit

The unit the values are given in, either in meters (`m`) or pixels (`px`). If no unit is given, uses the unit specified for the dimension or otherwise the default unit of the reference system.

Data type: **string**

Allowed values: px, m

`context = null`

Additional data to be passed to the process.

Any data type.

Data type: **any**

Return Value

A data cube with the newly computed values and the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*
- `DataCubePropertiesImmutable`
  Message: *The dimension properties (name, type, labels, reference system and resolution) must remain unchanged.*

Examples

Example #1

```
apply_neighborhood(data = $data, process = {"process_graph":{"udf":
{"process_id":"run_udf","arguments":{"data":
{"from_parameter":"data"},"udf":"ml.py","runtime":"Python"},"result":true}}}, size =
[{"dimension":"x","value":128,"unit":"px"},{"dimension":"y","value":128,"unit":"px"},
{"dimension":"t","value":"P5D"}], overlap = [{"dimension":"x","value":16,"unit":"px"},
{"dimension":"y","value":16,"unit":"px"},{"dimension":"t","value":"P3D"}])
```

See Also

- Apply explained in the openEO documentation

arccos

Inverse cosine

MATH > TRIGONOMETRIC

Download JSON

Description

`arccos(number|null x) : number|null`

Computes the arc cosine of `x`. The arc cosine is the inverse function of the cosine so that *`arccos(cos(x)) = x`*.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed angle in radians.

> Data type: **number, null**

Examples

Example #1

```
arccos(x = 1) => 0
```

See Also

- Inverse cosine explained by Wolfram MathWorld

arcosh

Inverse hyperbolic cosine

MATH > TRIGONOMETRIC

Download JSON

Description

`arcosh(number|null x) : number|null`

Computes the inverse hyperbolic cosine of `x`. It is the inverse function of the hyperbolic cosine so that *`arcosh(cosh(x)) = x`*.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed angle in radians.

> Data type: **number, null**

Examples

Example #1

```
arcosh(x = 1) => 0
```

See Also

- Inverse hyperbolic cosine explained by Wolfram MathWorld

arcsin

Inverse sine

MATH > TRIGONOMETRIC

Download JSON

Description

`arcsin(number|null x) : number|null`

Computes the arc sine of `x`. The arc sine is the inverse function of the sine so that *`arcsin(sin(x)) = x`*.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed angle in radians.

> Data type: **number, null**

Examples

Example #1

```
arcsin(x = 0) => 0
```

See Also

- Inverse sine explained by Wolfram MathWorld

arctan

Inverse tangent

MATH > TRIGONOMETRIC

Download JSON

Description

`arctan(number|null x) : number|null`

Computes the arc tangent of `x`. The arc tangent is the inverse function of the tangent so that *`arctan(tan(x)) = x`*.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

Data type: **number, null**

Return Value

The computed angle in radians.

Data type: **number, null**

Examples

Example #1

```
arctan(x = 0) => 0
```

See Also

- Inverse tangent explained by Wolfram MathWorld

arctan2

Inverse tangent of two numbers

MATH > TRIGONOMETRIC

Download JSON

Description

```
arctan2(number|null y, number|null x) : number|null
```

Computes the arc tangent of two numbers `x` and `y`. It is similar to calculating the arc tangent of *`y / x`*, except that the signs of both arguments are used to determine the quadrant of the result.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated if any of the arguments is `null`.

Parameters

y*

A number to be used as the dividend.

> Data type: **number, null**

x*

A number to be used as the divisor.

> Data type: **number, null**

Return Value

The computed angle in radians.

> Data type: **number, null**

Examples

Example #1

```
arctan2(y = 0, x = 0) => 0
```

Example #2

```
arctan2(y = null, x = 1.5) => null
```

See Also

- Two-argument inverse tangent explained by Wikipedia

ard_normalized_radar_backscatter

CARD4L compliant SAR NRB generation — **experimental**

CUBES SAR ARD

Download JSON

Description

```
ard_normalized_radar_backscatter(raster-cube data, ?collection-id:string|null elevation_model = null, ?boolean contributing_area = false, ?boolean ellipsoid_incidence_angle = false, ?boolean noise_removal = true, ?object options = {}) : raster-cube
```

Computes CARD4L compliant backscatter from SAR input. The radiometric correction coefficient is gamma0 (terrain), which is the ground area computed with terrain earth model in sensor line of sight.

Note that backscatter computation may require instrument specific metadata that is tightly coupled to the original SAR products. As a result, this process may only work in combination with loading data from specific collections, not with general data cubes.

This process uses bilinear interpolation, both for resampling the DEM and the backscatter.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

The source data cube containing SAR input.

Data type: **raster-cube**

`elevation_model` = `null`

The digital elevation model to use. Set to `null` (the default) to allow the back-end to choose, which will improve portability, but reduce reproducibility.

Data Types:

Data type: **collection-id:string**

> Data type: **null**

`contributing_area` = `false`

If set to `true`, a DEM-based local contributing area band named `contributing_area` is added. The values are given in square meters.

> Data type: **boolean**

`ellipsoid_incidence_angle` = `false`

If set to `true`, an ellipsoidal incidence angle band named `ellipsoid_incidence_angle` is added. The values are given in degrees.

> Data type: **boolean**

`noise_removal` = `true`

If set to `false`, no noise removal is applied. Defaults to `true`, which removes noise.

> Data type: **boolean**

`options` = `{}`

Proprietary options for the backscatter computations. Specifying proprietary options will reduce portability.

> Data type: **object**
>
> Each property: ✕ No

Return Value

Backscatter values expressed as gamma0 in linear scale.

In addition to the bands `contributing_area` and `ellipsoid_incidence_angle` that can optionally be added with corresponding parameters, the following bands are always added to the data cube:

- `mask`: A data mask that indicates which values are valid (1), invalid (0) or contain no-data (null).
- `local_incidence_angle`: A band with DEM-based local incidence angles in degrees.

The data returned is CARD4L compliant with corresponding metadata.

> Data type: **raster-cube**

Errors/Exceptions

- `DigitalElevationModelInvalid`
  Message: *The digital elevation model specified is either not a DEM or can't be used with the data cube given.*

See Also

- CEOS CARD4L specification
- Gamma nought (0) explained by EO4GEO body of knowledge.
- Reasoning behind the choice of bilinear resampling

ard_surface_reflectance

CARD4L compliant Surface Reflectance generation — **experimental**

CUBES OPTICAL ARD

Download JSON

Description

```
ard_surface_reflectance(raster-cube data, string atmospheric_correction_method, string
cloud_detection_method, ?collection-id:string|null elevation_model = null, ?object
atmospheric_correction_options = {}, ?object cloud_detection_options = {}) : raster-cube
```

Computes CARD4L compliant surface (bottom of atmosphere/top of canopy) reflectance values from optical input.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

data*

The source data cube containing multi-spectral optical top of the atmosphere (TOA) reflectances. There must be a single dimension of type `bands` available.

> Data type: **raster-cube**

atmospheric_correction_method*

The atmospheric correction method to use.

> Data type: **string**
>
> Allowed values: FORCE, iCOR

cloud_detection_method*

The cloud detection method to use.

Each method supports detecting different atmospheric disturbances such as clouds, cloud shadows, aerosols, haze, ozone and/or water vapour in optical imagery.

> Data type: **string**
>
> Allowed values: Fmask, s2cloudless, Sen2Cor

`elevation_model` = `null`

The digital elevation model to use. Set to `null` (the default) to allow the back-end to choose, which will improve portability, but reduce reproducibility.

Data Types:

> Data type: **collection-id:string**

> Data type: **null**

`atmospheric_correction_options` = `{}`

Proprietary options for the atmospheric correction method. Specifying proprietary options will reduce portability.

> Data type: **object**
>
> Each property: ✕ No

`cloud_detection_options` = `{}`

Proprietary options for the cloud detection method. Specifying proprietary options will reduce portability.

> Data type: **object**
>
> Each property: ✕ No

Return Value

Data cube containing bottom of atmosphere reflectances for each spectral band in the source data cube, with atmospheric disturbances like clouds and cloud shadows removed. No-data values (null) are directly set in the bands. Depending on the methods used, several additional bands will be added to the data cube:

Data cube containing bottom of atmosphere reflectances for each spectral band in the source data cube, with atmospheric disturbances like clouds and cloud shadows removed. Depending on the methods used, several additional bands will be added to the data cube:

- `date` (optional): Specifies per-pixel acquisition timestamps.
- `incomplete-testing` (required): Identifies pixels with a value of 1 for which the per-pixel tests (at least saturation, cloud and cloud shadows, see CARD4L specification for details) have not all been successfully completed. Otherwise, the value is 0.
- `saturation` (required) / `saturation_{band}` (optional): Indicates where pixels in the input spectral bands are saturated (1) or not (0). If the saturation is given per band, the band names are `saturation_{band}` with `{band}` being the band name from the source data cube.
- `cloud`, `shadow` (both required), `aerosol`, `haze`, `ozone`, `water_vapor` (all optional): Indicates the probability of pixels being an atmospheric disturbance such as clouds. All bands have values between 0 (clear) and 1, which describes the probability that it is an atmospheric disturbance.
- `snow-ice` (optional): Points to a file that indicates whether a pixel is assessed as being snow/ice (1) or not (0). All values describe the probability and must be between 0 and 1.
- `land-water` (optional): Indicates whether a pixel is assessed as being land (1) or water (0). All values describe the probability and must be between 0 and 1.
- `incidence-angle` (optional): Specifies per-pixel incidence angles in degrees.
- `azimuth` (optional): Specifies per-pixel azimuth angles in degrees.
- `sun-azimuth:` (optional): Specifies per-pixel sun azimuth angles in degrees.
- `sun-elevation` (optional): Specifies per-pixel sun elevation angles in degrees.

- `terrain-shadow` (optional): Indicates with a value of 1 whether a pixel is not directly illuminated due to terrain shadowing. Otherwise, the value is 0.
- `terrain-occlusion` (optional): Indicates with a value of 1 whether a pixel is not visible to the sensor due to terrain occlusion during off-nadir viewing. Otherwise, the value is 0.
- `terrain-illumination` (optional): Contains coefficients used for terrain illumination correction are provided for each pixel.

The data returned is CARD4L compliant with corresponding metadata.

> Data type: **raster-cube**

See Also

- CEOS CARD4L specification

array_append

Append a value to an array — **experimental**

ARRAYS

Download JSON

Description

```
array_append(array data, any value) : array
```

Appends a value to the end of the array. Array labels get discarded from the array.

> **Experimental**
> Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

`value`*

Value to append to the array.

Any data type is allowed.

Data type: **any**

Return Value

The new array with the value being appended.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

Examples

Example #1

```
array_append(data = [1,2], value = 3) => [1,2,3]
```

array_apply

Apply a process to each array element

ARRAYS

Download JSON

Description

```
array_apply(array data, process-graph:object process, ?any context = null) : array
```

Applies a process to each individual value in the array. This is basically what other languages call either a `for each` loop or a `map` function.

Parameters

`data`*

An array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

process*

A process that accepts and returns a single value and is applied on each individual value in the array. The process may consist of multiple sub-processes and could, for example, consist of processes such as `abs` or `linear_scale_range`.

Data type: **User-defined Process (process-graph:object)**

Parameters:

x*

The value of the current element being processed.

Any data type.

Data type: **any**

index*

The zero-based index of the current element being processed.

Data type: **integer**

Minimum value (inclusive): 0

label = null

The label of the current element being processed. Only populated for labeled arrays.

Data Types:

Data type: **number**

Data type: **string**

Data type: **null**

context = null

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new array.

Any data type.

Data type: **any**

`context = null`

Additional data to be passed to the process.

Any data type.

Data type: **any**

Return Value

An array with the newly computed values. The number of elements are the same as for the original array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

array_concat

Merge two arrays — **experimental**

ARRAYS

Download JSON

Description

```
array_concat(array array1, array array2) : array
```

Concatenates two arrays into a single array by appending the second array to the first array. Array labels get discarded from both arrays before merging.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`array1`*

The first array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`array2`*

The second array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

Return Value

The merged array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

Examples

Example #1

Concatenates two arrays containing different data type.

```
array_concat(array1 = ["a","b"], array2 = [1,2]) => ["a","b",1,2]
```

array_contains

Check whether the array contains a given value

ARRAYS COMPARISON REDUCER

Download JSON

Description

```
array_contains(array data, any value) : boolean
```

Checks whether the array specified for `data` contains the value specified in `value`. Returns `true` if there's a match, otherwise `false`.

Remarks:

- To get the index or the label of the value found, use `array_find`.
- All definitions for the process `eq` regarding the comparison of values apply here as well. A `null` return value from `eq` is handled exactly as `false` (no match).
- Data types MUST be checked strictly. For example, a string with the content *1* is not equal to the number *1*.
- An integer *1* is equal to a floating-point number *1.0* as `integer` is a sub-type of `number`. Still, this process may return unexpectedly `false` when comparing floating-point numbers due to floating-point inaccuracy in machine-based computation.
- Temporal strings are treated as normal strings and MUST NOT be interpreted.
- If the specified value is an array, object or null, the process always returns `false`. See the examples for one to check for `null` values.

Parameters

`data`*

List to find the value in.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

`value`*

Value to find in `data`.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if the list contains the value, false` otherwise.

> Data type: **boolean**

Examples

Example #1

```
array_contains(data = [1,2,3], value = 2) => true
```

Example #2

```
array_contains(data = ["A","B","C"], value = "b") => false
```

Example #3

```
array_contains(data = [1,2,3], value = "2") => false
```

Example #4

```
array_contains(data = [1,2,null], value = null) => true
```

Example #5

```
array_contains(data = [[1,2],[3,4]], value = [1,2]) => false
```

Example #6

```
array_contains(data = [[1,2],[3,4]], value = 2) => false
```

Example #7

```
array_contains(data = [{"a":"b"},{"c":"d"}], value = {"a":"b"}) => false
```

Processes

- Check for no-data values in arrays

array_create

Create an array — **experimental**

ARRAYS

Download JSON

Description

```
array_create(?array data = [], ?integer repeat = 1) : array
```

Creates a new array, which by default is empty.

The second parameter `repeat` allows to add the given array multiple times to the new array.

In most cases you can simply pass a (native) array to processes directly, but this process is especially useful to create a new array that is getting returned by a child process, for example in `apply_dimension`.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data = []`

A (native) array to fill the newly created array with. Defaults to an empty array.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

`repeat = 1`

The number of times the (native) array specified in `data` is repeatedly added after each other to the new array being created. Defaults to `1`.

> Data type: **integer**
>
> Minimum value (inclusive): 1

Return Value

The newly created array.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

Examples

Example #1

```
array_create() => []
```

Example #2

```
array_create(data = ["this","is","a","test"]) => ["this","is","a","test"]
```

Example #3

```
array_create(data = [null], repeat = 3) => [null,null,null]
```

Example #4

```
array_create(data = [1,2,3], repeat = 2) => [1,2,3,1,2,3]
```

array_create_labeled

Create a labeled array — **experimental**

ARRAYS

Download JSON

Description

```
array_create_labeled(any data, array<number|string> labels) : labeled-array
```

Creates a new labeled array by using the values from the `labels` array as labels and the values from the `data` array as the corresponding values.

The exception `ArrayLengthMismatch` is thrown, if the number of the values in the given arrays don't match exactly.

The primary use case here is to be able to transmit labeled arrays from the client to the server as JSON doesn't support this data type.

Experimental
Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array of values to be used.

> Any data type is allowed.
>
> Data type: **any**

`labels`*

An array of labels to be used.

> Data type: **array<number|string>**

Return Value

The newly created labeled array.

> Data type: **labeled-array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

Errors/Exceptions

- `ArrayLengthMismatch`
  Message: *The number of values in the parameters `data` and `labels` don't match.*

array_element

Get an element from an array

ARRAYS REDUCER

Download JSON

Description

```
array_element(array data, ?integer index, ?number|string label, ?boolean return_nodata = false) : any
```

Gives the element with the specified index or label from the array.

Either the parameter `index` or `label` must be specified, otherwise the `ArrayElementParameterMissing` exception is thrown. If both parameters are set the `ArrayElementParameterConflict` exception is thrown.

Parameters

`data`*

An array.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

`index`

The zero-based index of the element to retrieve.

> Data type: **integer**
>
> Minimum value (inclusive): 0

`label`

The label of the element to retrieve. Throws an `ArrayNotLabeled` exception, if the given array is not a labeled array and this parameter is set.

Data Types:

> Data type: **number**

> Data type: **string**

`return_nodata = false`

By default this process throws an `ArrayElementNotAvailable` exception if the index or label is invalid. If you want to return `null` instead, set this flag to `true`.

> Data type: **boolean**

Return Value

The value of the requested element.

> Any data type is allowed.
>
> Data type: **any**

Errors/Exceptions

- `ArrayElementNotAvailable`
  Message: *The array has no element with the specified index or label.*
- `ArrayElementParameterMissing`
  Message: *The process `array_element` requires either the `index` or `labels` parameter to be set.*
- `ArrayElementParameterConflict`
  Message: *The process `array_element` only allows that either the `index` or the `labels` parameter is set.*
- `ArrayNotLabeled`
  Message: *The array is not a labeled array, but the `label` parameter is set. Use the `index` instead.*

Examples

Example #1

```
array_element(data = [9,8,7,6,5], index = 2) => 7
```

Example #2

```
array_element(data = ["A","B","C"], index = 0) => "A"
```

Example #3

```
array_element(data = [], index = 0, return_nodata = true) => null
```

array_filter

Filter an array based on a condition

ARRAYS FILTER

Download JSON

Description

```
array_filter(array data, process-graph:object condition, ?any context = null) : array
```

Filters the array elements based on a logical expression so that afterwards an array is returned that only contains the values, indices and/or labels conforming to the condition.

Parameters

`data`*

An array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`condition`*

A condition that is evaluated against each value, index and/or label in the array. Only the array elements for which the condition returns `true` are preserved.

Data type: **User-defined Process (process-graph:object)**

Parameters:

`x`*

The value of the current element being processed.

Any data type.

Data type: **any**

`index`*

The zero-based index of the current element being processed.

Data type: **integer**

Minimum value (inclusive): 0

`label = null`

The label of the current element being processed. Only populated for labeled arrays.

Data Types:

Data type: **number**

Data type: **string**

Data type: **null**

`context = null`

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

`true` if the value should be kept in the array, otherwise `false`.

Data type: **boolean**

`context = null`

Additional data to be passed to the condition.

Any data type.

Data type: **any**

Return Value

An array filtered by the specified condition. The number of elements are less than or equal compared to the original array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

array_find

Get the index for a value in an array

Download JSON

Description

```
array_find(array data, any value, ?boolean reverse = false) : null|integer
```

Returns the zero-based index of the first (or last) occurrence of the value specified by `value` in the array specified by `data` or `null` if there is no match. Use the parameter `reverse` to switch from the first to the last match.

Remarks:

- Use `array_contains` to check if an array contains a value regardless of the position.
- Use `array_find_label` to find the index for a label.
- All definitions for the process `eq` regarding the comparison of values apply here as well. A `null` return value from `eq` is handled exactly as `false` (no match).
- Data types MUST be checked strictly. For example, a string with the content *1* is not equal to the number *1*.
- An integer *1* is equal to a floating-point number *1.0* as `integer` is a sub-type of `number`. Still, this process may return unexpectedly `false` when comparing floating-point numbers due to floating-point inaccuracy in machine-based computation.
- Temporal strings are treated as normal strings and MUST NOT be interpreted.
- If the specified value is an array, object or null, the process always returns `null`. See the examples for one to find `null` values.

Parameters

`data`*

List to find the value in.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

`value`*

Value to find in `data`.

> Any data type is allowed.
>
> Data type: **any**

`reverse = false`

By default, this process finds the index of the first match. To return the index of the last match instead, set this flag to `true`.

> Data type: **boolean**

Return Value

The index of the first element with the specified value. If no element was found, `null` is returned.

Data Types:

> Data type: **null**

> Data type: **integer**
>
> Minimum value (inclusive): 0

Examples

Example #1

```
array_find(data = [1,2,3,2,3], value = 2) => 1
```

Example #2

```
array_find(data = [1,2,3,2,3], value = 2, reverse = true) => 3
```

Example #3

```
array_find(data = ["A","B","C"], value = "b") => null
```

Example #4

```
array_find(data = [1,2,3], value = "2") => null
```

Example #5

```
array_find(data = [1,null,2,null], value = null) => 1
```

Example #6

```
array_find(data = [[1,2],[3,4]], value = [1,2]) => null
```

Example #7

```
array_find(data = [[1,2],[3,4]], value = 2) => null
```

Example #8

```
array_find(data = [{"a":"b"},{"c":"d"}], value = {"a":"b"}) => null
```

Processes

- Find no-data values in arrays

array_find_label

Get the index for a label in a labeled array — **experimental**

ARRAYS REDUCER

Download JSON

Description

`array_find_label(array data, number|string label) : null|integer`

Checks whether the labeled array specified for `data` has the label specified in `label` and returns the zero-based index for it. If there's no match as either the label doesn't exist or the array is not labeled, `null` is returned.

Use `array_find` to find the index for a given value in the array.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

List to find the label in.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`label`*

Label to find in `data`.

Data Types:

Data type: **number**

Data type: **string**

Return Value

The index of the element with the specified label assigned. If no such label was found, `null` is returned.

Data Types:

Data type: **null**

Data type: **integer**

Minimum value (inclusive): 0

array_interpolate_linear

One-dimensional linear interpolation for arrays — **experimental**

ARRAYS MATH MATH > INTERPOLATION

Download JSON

Description

```
array_interpolate_linear(array<number|null> data) : number|null
```

Performs a linear interpolation for each of the no-data values (`null`) in the array given, except for leading and trailing no-data values.

The linear interpolants are defined by the array indices or labels (x) and the values in the array (y).

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array of numbers and no-data values.

If the given array is a labeled array, the labels must have a natural/inherent label order and the process expects the labels to be sorted accordingly. This is the default behavior in openEO for spatial and temporal dimensions.

Data type: **array<number|null>**

Return Value

An array with no-data values being replaced with interpolated values. If not at least 2 numerical values are available in the array, the array stays the same.

Data type: **number, null**

Examples

Example #1

```
array_interpolate_linear(data = [null,1,null,6,null,-8]) => [null,1,3.5,6,-1,-8]
```

Example #2

```
array_interpolate_linear(data = [null,1,null,null]) => [null,1,null,null]
```

See Also

- Linear interpolation explained by Wikipedia

array_labels

Get the labels for an array

ARRAYS

Download JSON

Description

```
array_labels(array data) : array<number|string>
```

Gives all labels for a labeled array or gives all indices for an array without labels. If the array is not labeled, an array with the zero-based indices is returned. The labels or indices have the same order as in the array.

Parameters

data*

An array.

Data type: **array**

Return Value

The labels or indices as array.

Data type: **array<number|string>**

array_modify

Change the content of an array (remove, insert, update) — **experimental**

ARRAYS

Download JSON

Description

```
array_modify(array data, array values, integer index, ?integer length = 1) : array
```

Modify an array by removing, inserting or updating elements. Updating can be seen as removing elements followed by inserting new elements (not necessarily the same number).

All labels get discarded and the array indices are always a sequence of numbers with the step size of 1 and starting at 0.

> **Experimental**
> Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

The array to modify.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`values`*

The values to insert into the `data` array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`index`*

The index in the `data` array of the element to insert the value(s) before. If the index is greater than the number of elements in the `data` array, the process throws an `ArrayElementNotAvailable` exception.

To insert after the last element, there are two options:

1. Use the simpler processes `array_append` to append a single value or `array_concat` to append multiple values.
2. Specify the number of elements in the array. You can retrieve the number of elements with the process `count`, having the parameter `condition` set to `true`.

Data type: **integer**

Minimum value (inclusive): 0

`length = 1`

The number of elements in the `data` array to remove (or replace) starting from the given index. If the array contains fewer elements, the process simply removes all elements up to the end.

> Data type: **integer**
>
> Minimum value (inclusive): 0

Return Value

An array with values added, updated or removed.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

Errors/Exceptions

- `ArrayElementNotAvailable`

  Message: *The array can't be modified as the given index is larger than the number of elements in the array.*

Examples

Example #1

Replace a single value in the array.

```
array_modify(data = ["a","d","c"], values = ["b"], index = 1) => ["a","b","c"]
```

Example #2

Replace multiple values in the array.

```
array_modify(data = ["a","b",4,5], values = [1,2,3], index = 0, length = 2) =>
[1,2,3,4,5]
```

Example #3

Insert a value to the array at a given position.

```
array_modify(data = ["a","c"], values = ["b"], index = 1, length = 0) => ["a","b","c"]
```

Example #4

Remove a single value from the array.

```
array_modify(data = ["a","b",null,"c"], values = [], index = 2) => ["a","b","c"]
```

Example #5

Remove multiple values from the array.

```
array_modify(data = [null,null,"a","b","c"], values = [], index = 0, length = 2) =>
["a","b","c"]
```

Example #6

Remove multiple values from the end of the array and ignore that the given length is exceeding the size of the array.

```
array_modify(data = ["a","b","c"], values = [], index = 1, length = 10) => ["a"]
```

arsinh

Inverse hyperbolic sine

MATH > TRIGONOMETRIC

Download JSON

Description

`arsinh(number|null x) : number|null`

Computes the inverse hyperbolic sine of `x`. It is the inverse function of the hyperbolic sine so that *`arsinh(sinh(x)) = x`*.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

Data type: **number, null**

Return Value

The computed angle in radians.

Data type: **number, null**

Examples

Example #1

```
arsinh(x = 0) => 0
```

See Also

- Inverse hyperbolic sine explained by Wolfram MathWorld

artanh

Inverse hyperbolic tangent

MATH > TRIGONOMETRIC

Download JSON

Description

`artanh(number|null x) : number|null`

Computes the inverse hyperbolic tangent of `x`. It is the inverse function of the hyperbolic tangent so that *`artanh(tanh(x)) = x`*.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed angle in radians.

> Data type: **number, null**

Examples

Example #1

```
artanh(x = 0) => 0
```

See Also

- Inverse hyperbolic tangent explained by Wolfram MathWorld

atmospheric_correction

Apply atmospheric correction — **experimental**

CUBES OPTICAL

Download JSON

Description

```
atmospheric_correction(raster-cube data, string|null method, ?collection-id:string|null
elevation_model = null, ?object options = {}) : raster-cube
```

Applies an atmospheric correction that converts top of atmosphere reflectance values into bottom of atmosphere/top of canopy reflectance values.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

data*

Data cube containing multi-spectral optical top of atmosphere reflectances to be corrected.

> Data type: **raster-cube**

method*

The atmospheric correction method to use. To get reproducible results, you have to set a specific method.

Set to `null` to allow the back-end to choose, which will improve portability, but reduce reproducibility as you *may* get different results if you run the processes multiple times.

Data Types:

> Data type: **string**
>
> Allowed values: FORCE, iCOR

> Data type: **null**

elevation_model = null

The digital elevation model to use. Set to `null` (the default) to allow the back-end to choose, which will improve portability, but reduce reproducibility.

Data Types:

> Data type: **collection-id:string**

> Data type: **null**

`options = {}`

Proprietary options for the atmospheric correction method. Specifying proprietary options will reduce portability.

> Data type: **object**
>
> Each property: ✕ No

Return Value

Data cube containing bottom of atmosphere reflectances.

> Data type: **raster-cube**

Errors/Exceptions

- `DigitalElevationModelInvalid`
  Message: *The digital elevation model specified is either not a DEM or can't be used with the data cube given.*

See Also

- Atmospheric correction explained by EO4GEO body of knowledge.

between

Between comparison

COMPARISON

Download JSON

Description

```
between(any x, number|string min, number|string max, ?boolean exclude_max = false) :
boolean|null
```

By default, this process checks whether `x` is greater than or equal to `min` and lower than or equal to `max`, which is the same as computing `and(gte(x, min), lte(x, max))`. Therefore, all definitions from `and`, `gte` and `lte` apply here as well.

If `exclude_max` is set to `true` the upper bound is excluded so that the process checks whether `x` is greater than or equal to `min` and lower than `max`. In this case, the process works the same as computing `and(gte(x, min), lt(x, max))`.

Lower and upper bounds are not allowed to be swapped. So `min` MUST be lower than or equal to `max` or otherwise the process always returns `false`.

Parameters

x*

The value to check.

> Any data type is allowed.
>
> Data type: **any**

min*

Lower boundary (inclusive) to check against.

Data Types:

> Data type: **number**

> Data type: **date-time:string**

> Data type: **date:string**

> Data type: **time:string**

max*

Upper boundary (inclusive) to check against.

Data Types:

> Data type: **number**

> Data type: **date-time:string**

> Data type: **date:string**

> Data type: **time:string**

exclude_max = false

Exclude the upper boundary `max` if set to `true`. Defaults to `false`.

> Data type: **boolean**

Return Value

`true` if `x` is between the specified bounds, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
between(x = null, min = 0, max = 1) => null
```

Example #2

```
between(x = 1, min = 0, max = 1) => true
```

Example #3

```
between(x = 1, min = 0, max = 1, exclude_max = true) => false
```

Example #4

Swapped bounds (min is greater than max) MUST always return `false`.

```
between(x = 0.5, min = 1, max = 0) => false
```

Example #5

```
between(x = -0.5, min = -1, max = 0) => true
```

Example #6

```
between(x = "00:59:59Z", min = "01:00:00+01:00", max = "01:00:00Z") => true
```

Example #7

```
between(x = "2018-07-23T17:22:45Z", min = "2018-01-01T00:00:00Z", max = "2018-12-
31T23:59:59Z") => true
```

Example #8

```
between(x = "2000-01-01", min = "2018-01-01", max = "2020-01-01") => false
```

Example #9

```
between(x = "2018-12-31T17:22:45Z", min = "2018-01-01", max = "2018-12-31") => true
```

Example #10

```
between(x = "2018-12-31T17:22:45Z", min = "2018-01-01", max = "2018-12-31",
exclude_max = true) => false
```

ceil

Round fractions up

MATH > ROUNDING

Download JSON

Description

```
ceil(number|null x) : integer|null
```

The least integer greater than or equal to the number `x`.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number to round up.

> Data type: **number, null**

Return Value

The number rounded up.

> Data type: **integer, null**

Examples

Example #1

```
ceil(x = 0) => 0
```

Example #2

```
ceil(x = 3.5) => 4
```

Example #3

```
ceil(x = -0.4) => 0
```

Example #4

```
ceil(x = -3.5) => -3
```

See Also

- Ceiling explained by Wolfram MathWorld

climatological_normal

Compute climatology normals

CLIMATOLOGY

Download JSON

Description

```
climatological_normal(raster-cube data, string period, ?temporal-interval:array<year:string>
climatology_period = ["1981","2010"]) : raster-cube
```

Climatological normal period is a usually 30 year average of a weather variable. Climatological normals are used as an average or baseline to evaluate climate events and provide context for yearly, monthly, daily or seasonal variability. The default climatology period is from 1981 until 2010 (both inclusive).

Parameters

`data`*

A data cube with exactly one temporal dimension. The data cube must span at least the temporal interval specified in the parameter `climatology-period`.

Seasonal periods may span two consecutive years, e.g. temporal winter that includes months December, January and February. If the required months before the actual climate period are available, the season is taken into account. If not available, the first season is not taken into account and the seasonal mean is based on one year less than the other seasonal normals. The incomplete season at the end of the last year is never taken into account.

> Data type: **raster-cube**

`period`*

The time intervals to aggregate the average value for. The following pre-defined frequencies are supported:

- `day`: Day of the year
- `month`: Month of the year
- `climatology-period`: The period specified in the `climatology-period`.
- `season`: Three month periods of the calendar seasons (December - February, March - May, June - August, September - November).
- `tropical-season`: Six month periods of the tropical seasons (November - April, May - October).

> Data type: **string**
>
> Allowed values: day, month, season, tropical-season, climatology-period

`climatology_period = ["1981","2010"]`

The climatology period as a closed temporal interval. The first element of the array is the first year to be fully included in the temporal interval. The second element is the last year to be fully included in the temporal interval. The default period is from 1981 until 2010 (both inclusive).

Data type:	**temporal-interval:array<year:string>**
Min. number of items:	2
Max. number of items:	2
Array items:	Data type: **year:string** Min Length: 4 Max Length: 4 Pattern: `^\d{4}$`

Return Value

A data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged, except for the resolution and dimension labels of the temporal dimension. The temporal dimension has the following dimension labels:

- `day`: `001` - `365`
- `month`: `01` - `12`
- `climatology-period`: `climatology-period`
- `season`: `djf` (December - February), `mam` (March - May), `jja` (June - August), `son` (September - November)
- `tropical-season`: `ndjfma` (November - April), `mjjaso` (May - October)

Data type: **raster-cube**

See Also

- Background information on climatology normal by Wikipedia

clip

Clip a value between a minimum and a maximum

MATH

Download JSON

Description

```
clip(number|null x, number min, number max) : number|null
```

Clips a number between specified minimum and maximum values. A value larger than the maximum value is set to the maximum value, a value lower than the minimum value is set to the minimum value.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

min*

Minimum value. If the value is lower than this value, the process will return the value of this parameter.

> Data type: **number**

max*

Maximum value. If the value is greater than this value, the process will return the value of this parameter.

> Data type: **number**

Return Value

The value clipped to the specified range.

> Data type: **number, null**

Examples

Example #1

```
clip(x = -5, min = -1, max = 1) => -1
```

Example #2

```
clip(x = 10.001, min = 1, max = 10) => 10
```

Example #3

```
clip(x = 0.000001, min = 0, max = 0.02) => 0.000001
```

Example #4

```
clip(x = null, min = 0, max = 1) => null
```

cloud_detection

Create cloud masks — **experimental**

CUBES OPTICAL

Download JSON

Description

```
cloud_detection(raster-cube data, string|null method, ?object options = {}) : raster-cube
```

Detects atmospheric disturbances such as clouds, cloud shadows, aerosols, haze, ozone and/or water vapour in optical imagery.

It creates a data cube with the spatial and temporal dimensions compatible to the source data cube and a dimension that contains a dimension label for each of the supported/considered atmospheric disturbances. The naming of the bands follows these pre-defined values:

- `cloud`
- `shadow`
- `aerosol`
- `haze`
- `ozone`
- `water_vapor`

All bands have values between 0 (clear) and 1, which describes the probability that it is an atmospheric disturbance.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

The source data cube containing multi-spectral optical top of the atmosphere (TOA) reflectances on which to perform cloud detection.

Data type: **raster-cube**

`method`*

The cloud detection method to use. To get reproducible results, you have to set a specific method.

Set to `null` to allow the back-end to choose, which will improve portability, but reduce reproducibility as you *may* get different results if you run the processes multiple times.

Data Types:

Data type: **string**

Allowed values: Fmask, s2cloudless, Sen2Cor

Data type: **null**

`options = {}`

Proprietary options for the cloud detection method. Specifying proprietary options will reduce portability.

Data type: **object**

> Each property: ✕ No

Return Value

A data cube with bands for the atmospheric disturbances. Each of the masks contains values between 0 and 1. The data cube has the same spatial and temporal dimensions as the source data cube and a dimension that contains a dimension label for each of the supported/considered atmospheric disturbance.

> Data type: **raster-cube**

See Also

- Cloud mask explained by EO4GEO body of knowledge.

constant

Define a constant value

MATH > CONSTANTS

Download JSON

Description

```
constant(any x) : any
```

Defines a constant value that can be reused in multiple places of a process.

Parameters

x*

The value of the constant.

> Any data type.
>
> Data type: **any**

Return Value

The value of the constant.

> Any data type.
>
> Data type: **any**

cos

Cosine

MATH > TRIGONOMETRIC

Download JSON

Description

`cos(number|null x) : number|null`

Computes the cosine of `x`.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

An angle in radians.

> Data type: **number, null**

Return Value

The computed cosine of `x`.

> Data type: **number, null**

Examples

Example #1

```
cos(x = 0) => 1
```

See Also

- Cosine explained by Wolfram MathWorld

cosh

Hyperbolic cosine

MATH > TRIGONOMETRIC

Download JSON

Description

`cosh(number|null x) : number|null`

Computes the hyperbolic cosine of `x`.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

An angle in radians.

> Data type: **number, null**

Return Value

The computed hyperbolic cosine of `x`.

> Data type: **number, null**

Examples

Example #1

```
cosh(x = 0) => 1
```

See Also

- Hyperbolic cosine explained by Wolfram MathWorld

count

Count the number of elements

ARRAYS MATH > STATISTICS REDUCER

Download JSON

Description

`count(array data, ?process-graph:object|boolean|null condition = null, ?any context = null) : number`

Gives the number of elements in an array that matches the specified condition.

Remarks:

- Counts the number of valid elements by default (`condition` is set to `null`). A valid element is every element for which `is_valid` returns `true`.
- To count all elements in a list set the `condition` parameter to boolean `true`.

Parameters

`data`*

An array with elements of any data type.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`condition` = `null`

A condition consists of one or more processes, which in the end return a boolean value. It is evaluated against each element in the array. An element is counted only if the condition returns `true`. Defaults to count valid elements in a list (see `is_valid`). Setting this parameter to boolean `true` counts all elements in the list.

Data Types:

Data type: **User-defined Process (process-graph:object)**

Parameters:

`x`*

The value of the current element being processed.

Any data type.

Data type: **any**

`context` = `null`

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

`true` if the element should increase the counter, otherwise `false`.

Data type: **boolean**

All elements

> Boolean `true` counts all elements in the list.
>
> Data type: **boolean**
>
> Allowed value: true

> **Valid elements**
>
> `null` counts valid elements in the list.
>
> Data type: **null**

`context = null`

Additional data to be passed to the condition.

> Any data type.
>
> Data type: **any**

Return Value

The counted number of elements.

> Data type: **number**

Examples

Example #1

```
count(data = []) => 0
```

Example #2

```
count(data = [1,0,3,2]) => 4
```

Example #3

```
count(data = ["ABC",null]) => 1
```

Example #4

```
count(data = [false,null], condition = true) => 2
```

Example #5

```
count(data = [0,1,2,3,4,5,null], condition = {"gt":{"process_id":"gt","arguments":
{"x":{"from_parameter":"element"},"y":2},"result":true}}) => 3
```

create_raster_cube

Create an empty raster data cube

CUBES

Download JSON

Description

`create_raster_cube() : raster-cube`

Creates a new raster data cube without dimensions. Dimensions can be added with `add_dimension`.

Parameters

This process has no parameters.

Return Value

An empty raster data cube with zero dimensions.

> Data type: **raster-cube**

See Also

- Data Cubes explained in the openEO documentation

cummax

Cumulative maxima — **experimental**

MATH > CUMULATIVE

Download JSON

Description

`cummax(array<number|null> data, ?boolean ignore_nodata = true) : array<number|null>`

Finds cumulative maxima of an array of numbers. Every computed element is equal to the bigger one between the current element and the previously computed element. The returned array and the input array have always the same length.

By default, no-data values are skipped, but stay in the result. Setting the `ignore_nodata` flag to `true` makes that once a no-data value (`null`) is reached all following elements are set to `null` in the result.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not and ignores them by default. Setting this flag to `false` considers no-data values so that `null` is set for all the following elements.

> Data type: **boolean**

Return Value

An array with the computed cumulative maxima.

> Data type: **array<number|null>**

Examples

Example #1

```
cummax(data = [1,3,5,3,1]) => [1,3,5,5,5]
```

Example #2

```
cummax(data = [1,3,null,5,1]) => [1,3,null,5,5]
```

Example #3

```
cummax(data = [1,3,null,5,1], ignore_nodata = false) => [1,3,null,null,null]
```

cummin

Cummulative minima — **experimental**

MATH > CUMULATIVE

Download JSON

Description

```
cummin(array<number|null> data, ?boolean ignore_nodata = true) : array<number|null>
```

Finds cumulative minima of an array of numbers. Every computed element is equal to the smaller one between the current element and the previously computed element. The returned array and the input array have always the same length.

By default, no-data values are skipped, but stay in the result. Setting the `ignore_nodata` flag to `true` makes that once a no-data value (`null`) is reached all following elements are set to `null` in the result.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not and ignores them by default. Setting this flag to `false` considers no-data values so that `null` is set for all the following elements.

> Data type: **boolean**

Return Value

An array with the computed cumulative minima.

> Data type: **array<number|null>**

Examples

Example #1

```
cummin(data = [5,3,1,3,5]) => [5,3,1,1,1]
```

Example #2

```
cummin(data = [5,3,null,1,5]) => [5,3,null,1,1]
```

Example #3

```
cummin(data = [5,3,null,1,5], ignore_nodata = false) => [5,3,null,null,null]
```

cumproduct

Cumulative products — **experimental**

MATH > CUMULATIVE

Download JSON

Description

```
cumproduct(array<number|null> data, ?boolean ignore_nodata = true) : array<number|null>
```

Computes cumulative products of an array of numbers. Every computed element is equal to the product of the current and all previous values. The returned array and the input array have always the same length.

By default, no-data values are skipped, but stay in the result. Setting the `ignore_nodata` flag to `true` makes that once a no-data value (`null`) is reached all following elements are set to `null` in the result.

> **Experimental**
> Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not and ignores them by default. Setting this flag to `false` considers no-data values so that `null` is set for all the following elements.

> Data type: **boolean**

Return Value

An array with the computed cumulative products.

> Data type: **array<number|null>**

Examples

Example #1

```
cumproduct(data = [1,3,5,3,1]) => [1,3,15,45,45]
```

Example #2

```
cumproduct(data = [1,2,3,null,3,1]) => [1,2,6,null,18,18]
```

Example #3

```
cumproduct(data = [1,2,3,null,3,1], ignore_nodata = false) => [1,2,6,null,null,null]
```

cumsum

Cumulative sums — **experimental**

MATH > CUMULATIVE

Download JSON

Description

```
cumsum(array<number|null> data, ?boolean ignore_nodata = true) : array<number|null>
```

Computes cumulative sums of an array of numbers. Every computed element is equal to the sum of current and all previous values. The returned array and the input array have always the same length.

By default, no-data values are skipped, but stay in the result. Setting the `ignore_nodata` flag to `true` makes that once a no-data value (`null`) is reached all following elements are set to `null` in the result.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

An array of numbers.

Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not and ignores them by default. Setting this flag to `false` considers no-data values so that `null` is set for all the following elements.

Data type: **boolean**

Return Value

An array with the computed cumulative sums.

Data type: **array<number|null>**

Examples

Example #1

```
cumsum(data = [1,3,5,3,1]) => [1,4,9,12,13]
```

Example #2

```
cumsum(data = [1,3,null,3,1]) => [1,4,null,7,8]
```

Example #3

```
cumsum(data = [1,3,null,3,1], ignore_nodata = false) => [1,4,null,null,null]
```

date_shift

Manipulates dates and times by addition or subtraction — **experimental**

DATE & TIME

Download JSON

Description

```
date_shift(string date, integer value, string unit) : date-time:string|date:string
```

Based on a given date (and optionally time), calculates a new date (and time if given) by adding or subtracting a given temporal period.

Some specifics about dates and times need to be taken into account:

- This process doesn't have any effect on the time zone.
- It doesn't take daylight saving time (DST) into account as only dates and time in UTC (with potential numerical time zone modifier) are supported.
- Leap years are implemented in a way that computations handle them gracefully (see parameter `unit` for details).
- Leap seconds are mostly ignored in manipulations as they don't follow a regular pattern. Leap seconds can be passed to the process, but will never be returned.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`date`*

The date (and optionally time) to manipulate.

If the given date doesn't include the time, the process assumes that the time component is `00:00:00Z` (i.e. midnight, in UTC). The millisecond part of the time is optional and defaults to `0` if not given.

Data Types:

> Data type: **date-time:string**

> Data type: **date:string**

`value`*

The period of time in the unit given that is added (positive numbers) or subtracted (negative numbers). The value `0` doesn't have any effect.

> Data type: **integer**

`unit`*

The unit for the value given. The following pre-defined units are available:

- millisecond: Milliseconds
- second: Seconds - leap seconds are ignored in computations.
- minute: Minutes
- hour: Hours
- day: Days - changes only the the day part of a date
- week: Weeks (equivalent to 7 days)
- month: Months
- year: Years

Manipulations with the unit `year`, `month`, `week` or `day` do never change the time. If any of the manipulations result in an invalid date or time, the corresponding part is rounded down to the next valid date or time respectively. For example, adding a month to `2020-01-31` would result in `2020-02-29`.

> Data type: **string**
>
> Allowed values: millisecond, second, minute, hour, day, week, month, year

Return Value

The manipulated date. If a time component was given in the parameter `date`, the time component is returned with the date.

Data Types:

> Data type: **date-time:string**

> Data type: **date:string**

Examples

Example #1

```
date_shift(date = "2020-02-01T17:22:45Z", value = 6, unit = "month") => "2020-08-01T17:22:45Z"
```

Example #2

```
date_shift(date = "2021-03-31T00:00:00+02:00", value = -7, unit = "day") => "2021-03-24T00:00:00+02:00"
```

Example #3

Adding a year to February 29th in a leap year will result in February 28th in the next (non-leap) year.

```
date_shift(date = "2020-02-29T17:22:45Z", value = 1, unit = "year") => "2021-02-28T17:22:45Z"
```

Example #4

Adding a month to January 31th will result in February 29th in leap years.

```
date_shift(date = "2020-01-31", value = 1, unit = "month") => "2020-02-29"
```

Example #5

The process skips over the leap second `2016-12-31T23:59:60Z`.

```
date_shift(date = "2016-12-31T23:59:59Z", value = 1, unit = "second") => "2017-01-01T00:00:00Z"
```

Example #6

Milliseconds can be added or subtracted. If not given, the default value is `0`.

```
date_shift(date = "2018-12-31T17:22:45Z", value = 1150, unit = "millisecond") => "2018-12-31T17:22:46.150Z"
```

Example #7

```
date_shift(date = "2018-01-01", value = 25, unit = "hour") => "2018-01-02"
```

Example #8

```
date_shift(date = "2018-01-01", value = -1, unit = "hour") => "2017-12-31"
```

dimension_labels

Get the dimension labels

CUBES

Download JSON

Description

`dimension_labels(raster-cube data, string dimension) : array<number|string>`

Gives all labels for a dimension in the data cube. The labels have the same order as in the data cube.

If a dimension with the specified name does not exist, the process fails with a `DimensionNotAvailable` exception.

Parameters

`data`*

The data cube.

> Data type: **raster-cube**

`dimension`*

The name of the dimension to get the labels for.

> Data type: **string**

Return Value

The labels as an array.

> Data type: **array<number|string>**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

divide

Division of two numbers

MATH

Download JSON

Description

`divide(number|null x, number|null y) : number|null`

Divides argument `x` by the argument `y` (*`x / y`*) and returns the computed result.

No-data values are taken into account so that `null` is returned if any element is such a value.

The computations follow IEEE Standard 754 whenever the processing environment supports it. Therefore, a division by zero results in ±infinity if the processing environment supports it. Otherwise, a `DivisionByZero` exception must the thrown.

Parameters

x*

The dividend.

> Data type: **number, null**

y*

The divisor.

> Data type: **number, null**

Return Value

The computed result.

> Data type: **number, null**

Errors/Exceptions

- `DivisionByZero`
  Message: *Division by zero is not supported.*

Examples

Example #1

```
divide(x = 5, y = 2.5) => 2
```

Example #2

```
divide(x = -2, y = 4) => -0.5
```

Example #3

```
divide(x = 1, y = null) => null
```

See Also

- Division explained by Wolfram MathWorld
- IEEE Standard 754-2019 for Floating-Point Arithmetic

drop_dimension

Remove a dimension

CUBES

Download JSON

Description

```
drop_dimension(raster-cube data, string name) : raster-cube
```

Drops a dimension from the data cube.

Dropping a dimension only works on dimensions with a single dimension label left, otherwise the process fails with a `DimensionLabelCountMismatch` exception. Dimension values can be reduced to a single value with a filter such as `filter_bands` or the `reduce_dimension` process. If a dimension with the specified name does not exist, the process fails with a `DimensionNotAvailable` exception.

Parameters

`data`*

The data cube to drop a dimension from.

> Data type: **raster-cube**

`name`*

Name of the dimension to drop.

> Data type: **string**

Return Value

A data cube without the specified dimension. The number of dimensions decreases by one, but the dimension properties (name, type, labels, reference system and resolution) for all other dimensions remain unchanged.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionLabelCountMismatch`
  Message: *The number of dimension labels exceeds one, which requires a reducer.*
- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

e

Euler's number (e)

MATH > CONSTANTS MATH > EXPONENTIAL & LOGARITHMIC

Download JSON

Description

```
e() : number
```

The real number *e* is a mathematical constant that is the base of the natural logarithm such that *`ln(e) = 1`*. The numerical value is approximately *2.71828*.

Parameters

This process has no parameters.

Return Value

The numerical value of Euler's number.

> Data type: **number**

See Also

- Mathematical constant e explained by Wolfram MathWorld

eq

Equal to comparison

TEXTS COMPARISON

Download JSON

Description

```
eq(any x, any y, ?number|null delta = null, ?boolean case_sensitive = true) : boolean|null
```

Compares whether `x` is strictly equal to `y`.

Remarks:

- Data types MUST be checked strictly. For example, a string with the content *1* is not equal to the number *1*. Nevertheless, an integer *1* is equal to a floating-point number *1.0* as `integer` is a sub-type of `number`.
- If any operand is `null`, the return value is `null`. Therefore, `eq(null, null)` returns `null` instead of `true`.
- If any operand is an array or object, the return value is `false`.
- Strings are expected to be encoded in UTF-8 by default.
- Temporal strings MUST be compared differently than other strings and MUST NOT be compared based on their string representation due to different possible representations. For example, the time zone representation `Z` (for UTC) has the same meaning as `+00:00`.

Parameters

x*

First operand.

> Any data type is allowed.
>
> Data type: **any**

y*

Second operand.

> Any data type is allowed.
>
> Data type: **any**

`delta = null`

Only applicable for comparing two numbers. If this optional parameter is set to a positive non-zero number the equality of two numbers is checked against a delta value. This is especially useful to circumvent problems with floating-point inaccuracy in machine-based computation.

This option is basically an alias for the following computation: `lte(abs(minus([x, y]), delta)`

> Data type: **number, null**

`case_sensitive = true`

Only applicable for comparing two strings. Case sensitive comparison can be disabled by setting this parameter to `false`.

> Data type: **boolean**

Return Value

`true` if `x` is equal to `y`, `null` if any operand is `null`, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
eq(x = 1, y = null) => null
```

Example #2

```
eq(x = null, y = null) => null
```

Example #3

```
eq(x = 1, y = 1) => true
```

Example #4

```
eq(x = 1, y = "1") => false
```

Example #5

```
eq(x = 0, y = false) => false
```

Example #6

```
eq(x = 1.02, y = 1, delta = 0.01) => false
```

Example #7

```
eq(x = -1, y = -1.001, delta = 0.01) => true
```

Example #8

```
eq(x = 115, y = 110, delta = 10) => true
```

Example #9

```
eq(x = "Test", y = "test") => false
```

Example #10

```
eq(x = "Test", y = "test", case_sensitive = false) => true
```

Example #11

```
eq(x = "Ä", y = "ä", case_sensitive = false) => true
```

Example #12

```
eq(x = "00:00:00+00:00", y = "00:00:00Z") => true
```

Example #13

y is not a valid date-time representation and therefore will be treated as a string so that the provided values are not equal.

```
eq(x = "2018-01-01T12:00:00Z", y = "2018-01-01T12:00:00") => false
```

Example #14

01:00 in the time zone +1 is equal to 00:00 in UTC.

```
eq(x = "2018-01-01T00:00:00Z", y = "2018-01-01T01:00:00+01:00") => true
```

Example #15

```
eq(x = [1,2,3], y = [1,2,3]) => false
```

exp

Exponentiation to the base e

MATH > EXPONENTIAL & LOGARITHMIC

Download JSON

Description

`exp(number|null p) : number|null`

Exponential function to the base *e* raised to the power of `p`.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

p*

The numerical exponent.

> Data type: **number, null**

Return Value

The computed value for *e* raised to the power of `p`.

> Data type: **number, null**

Examples

Example #1

```
exp(p = 0) => 1
```

Example #2

```
exp(p = null) => null
```

See Also

- Exponential function explained by Wolfram MathWorld

extrema

Minimum and maximum values

Download JSON

Description

```
extrema(array<number|null> data, ?boolean ignore_nodata = true) : array<number>|array<null>
```

Two element array containing the minimum and the maximum values of `data`.

This process is basically an alias for calling both `min` and `max`, but may be implemented more performant by back-ends as it only needs to iterate over the data once instead of twice.

Parameters

`data`*

An array of numbers.

Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that an array with two `null` values is returned if any value is such a value.

Data type: **boolean**

Return Value

An array containing the minimum and maximum values for the specified numbers. The first element is the minimum, the second element is the maximum. If the input array is empty both elements are set to `null`.

Data Types:

Data type: **array<number>**

Min. number of items: 2

Max. number of items: 2

Array items:

Data type: **number**

Data type: **array<null>**

Min. number of items: 2

Max. number of items: 2

Array items:

Data type: **null**

Examples

Example #1

```
extrema(data = [1,0,3,2]) => [0,3]
```

Example #2

```
extrema(data = [5,2.5,null,-0.7]) => [-0.7,5]
```

Example #3

```
extrema(data = [1,0,3,null,2], ignore_nodata = false) => [null,null]
```

Example #4

The input array is empty: return two `null` values.

```
extrema(data = []) => [null,null]
```

filter_bands

Filter the bands by names

CUBES FILTER

Download JSON

Description

```
filter_bands(raster-cube data, ?array<band-name:string> bands = [], ?array<array<number>>
wavelengths = []) : raster-cube
```

Filters the bands in the data cube so that bands that don't match any of the criteria are dropped from the data cube. The data cube is expected to have only one dimension of type `bands`. Fails with a `DimensionMissing` exception if no such dimension exists.

The following criteria can be used to select bands:

- `bands`: band name or common band name (e.g. `B01`, `B8A`, `red` or `nir`)
- `wavelengths`: ranges of wavelengths in micrometers (μm) (e.g. 0.5 - 0.6)

All these information are exposed in the band metadata of the collection. To keep algorithms interoperable it is recommended to prefer the common band names or the wavelengths over band names that are specific to the collection and/or back-end.

If multiple criteria are specified, any of them must match and not all of them, i.e. they are combined with an OR-operation. If no criteria are specified, the `BandFilterParameterMissing` exception must be thrown.

Important: The order of the specified array defines the order of the bands in the data cube, which can be important for subsequent processes. If multiple bands are matched by a single criterion (e.g. a range of wavelengths), they stay in the original order.

Parameters

`data`*

A data cube with bands.

> Data type: **raster-cube**

`bands` = [ ]

A list of band names. Either the unique band name (metadata field `name` in bands) or one of the common band names (metadata field `common_name` in bands). If the unique band name and the common name conflict, the unique band name has a higher priority.

The order of the specified array defines the order of the bands in the data cube. If multiple bands match a common name, all matched bands are included in the original order.

> Data type: **array<band-name:string>**
>
> Array items:
> > Data type: **band-name:string**

`wavelengths` = [ ]

A list of sub-lists with each sub-list consisting of two elements. The first element is the minimum wavelength and the second element is the maximum wavelength. Wavelengths are specified in micrometers (µm).

The order of the specified array defines the order of the bands in the data cube. If multiple bands match the wavelengths, all matched bands are included in the original order.

> Data type: **array<array<number>>**
>
> Array items:
> > Data type: **array<number>**
> >
> > Min. number of items: 2
> >
> > Max. number of items: 2
> >
> > Array items:
> > > Data type: **number**
> >
> > Examples: `[ [ 0.45, 0.5 ], [ 0.6, 0.7 ] ]`

Return Value

A data cube limited to a subset of its original bands. The dimensions and dimension properties (name, type, labels, reference system and resolution) remain unchanged, except that the dimension of type `bands` has less (or the same) dimension labels.

> Data type: **raster-cube**

Errors/Exceptions

- `BandFilterParameterMissing`
  Message: *The process `filter_bands` requires any of the parameters `bands`, `common_names` or `wavelengths` to be set.*
- `DimensionMissing`
  Message: *A band dimension is missing.*

See Also

- Filters explained in the openEO documentation
- List of common band names as specified by the STAC specification

filter_bbox

Spatial filter using a bounding box

CUBES FILTER

Download JSON

Description

`filter_bbox(raster-cube data, bounding-box:object extent) : raster-cube`

Limits the data cube to the specified bounding box.

The filter retains a pixel in the data cube if the point at the pixel center intersects with the bounding box (as defined in the Simple Features standard by the OGC).

Parameters

data*

A data cube.

Data type: **raster-cube**

extent*

A bounding box, which may include a vertical axis (see `base` and `height`).

Data type: **bounding-box:object**

Object Properties:

west** * West (lower left corner, coordinate axis 1).

Data type: **number**

south** * South (lower left corner, coordinate axis 2).

Data type: **number**

east** * East (upper right corner, coordinate axis 1).

Data type: **number**

north * North (upper right corner, coordinate axis 2).

Data type: **number**

base Base (optional, lower left corner, coordinate axis 3).

Data type: **number, null**

Default value: `null`

height Height (optional, upper right corner, coordinate axis 3).

Data type: **number, null**

Default value: `null`

crs Coordinate reference system of the extent, specified as as EPSG code, WKT2 (ISO 19162) string or PROJ definition (deprecated). Defaults to `4326` (EPSG code 4326) unless the client explicitly requests a different coordinate reference system.

Data type: **any**

Default value: 4326

Return Value

A data cube restricted to the bounding box. The dimensions and dimension properties (name, type, labels, reference system and resolution) remain unchanged, except that the spatial dimensions have less (or the same) dimension labels.

Data type: **raster-cube**

See Also

- Filters explained in the openEO documentation
- Official EPSG code registry
- PROJ parameters for cartographic projections
- Simple Features standard by the OGC
- Unofficial EPSG code database

filter_labels

Filter dimension labels based on a condition — **experimental**

CUBES FILTER

Download JSON

Description

```
filter_labels(raster-cube data, process-graph:object condition, string dimension, ?any context = null) : raster-cube
```

Filters the dimension labels in the data cube for the given dimension. Only the dimension labels that match the specified condition are preserved, all other labels with their corresponding data get removed.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

data*

A data cube.

> Data type: **raster-cube**

condition*

A condition that is evaluated against each dimension label in the specified dimension. A dimension label and the corresponding data is preserved for the given dimension, if the condition returns `true`.

> Data type: **User-defined Process (process-graph:object)**
>
> **Parameters:**
>
> #### value*
>
> A single dimension label to compare against. The data type of the parameter depends on the dimension labels set for the dimension.
>
> **Data Types:**
>
> > Data type: **number**
>
> > Data type: **string**
>
> #### context = null
>
> Additional data passed by the user.
>
> > Any data type.
>
> > Data type: **any**
>
> **Expected Return Value:**
>
> `true` if the dimension label should be kept in the data cube, otherwise `false`.

> Data type: **boolean**

`dimension`*

The name of the dimension to filter on. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

> Data type: **string**

`context` = `null`

Additional data to be passed to the condition.

> Any data type.
>
> Data type: **any**

Return Value

A data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged, except that the given dimension has less (or the same) dimension labels.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

Examples

Example #1

Filters the data cube to only contain data from platform Sentinel-2A. This example assumes that the data cube has a dimension `platform` so that computations can distinguish between Sentinel-2A and Sentinel-2B data.

```
filter_labels(data = $sentinel2_data, condition = {"process_graph":{"eq":
{"process_id":"eq","arguments":{"x":{"from_parameter":"value"},"y":"Sentinel-
2A","case_sensitive":false},"result":true}}}, dimension = "platform")
```

See Also

- Filters explained in the openEO documentation

filter_spatial

Spatial filter using geometries

CUBES FILTER

Download JSON

Description

```
filter_spatial(raster-cube data, geojson:object geometries) : raster-cube
```

Limits the data cube over the spatial dimensions to the specified geometries.

- For **polygons**, the filter retains a pixel in the data cube if the point at the pixel center intersects with at least one of the polygons (as defined in the Simple Features standard by the OGC).
- For **points**, the process considers the closest pixel center.
- For **lines** (line strings), the process considers all the pixels whose centers are closest to at least one point on the line.

More specifically, pixels outside of the bounding box of the given geometry will not be available after filtering. All pixels inside the bounding box that are not retained will be set to `null` (no data).

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`geometries`*

One or more geometries used for filtering, specified as GeoJSON.

> Data type: **geojson:object**

Return Value

A data cube restricted to the specified geometries. The dimensions and dimension properties (name, type, labels, reference system and resolution) remain unchanged, except that the spatial dimensions have less (or the same) dimension labels.

> Data type: **raster-cube**

See Also

- Filters explained in the openEO documentation
- Simple Features standard by the OGC

filter_temporal

Temporal filter based on temporal intervals

CUBES FILTER

Download JSON

Description

```
filter_temporal(raster-cube data, temporal-interval:array<string|null> extent, ?string|null dimension = null) : raster-cube
```

Limits the data cube to the specified interval of dates and/or times.

More precisely, the filter checks whether each of the temporal dimension labels is greater than or equal to the lower boundary (start date/time) and less than the value of the upper boundary (end date/time). This corresponds to a left-closed interval, which contains the lower boundary but not the upper boundary.

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`extent`*

Left-closed temporal interval, i.e. an array with exactly two elements:

1. The first element is the start of the temporal interval. The specified instance in time is **included** in the interval.
2. The second element is the end of the temporal interval. The specified instance in time is **excluded** from the interval.

The specified temporal strings follow RFC 3339. Also supports open intervals by setting one of the boundaries to `null`, but never both.

> Data type: **temporal-interval:array<date-time:string|date:string|year:string|null>**
>
> Min. number of items: 2
>
> Max. number of items: 2
>
> Array items: Data type: **any**
>
> Examples:
> - `[ "2015-01-01T00:00:00Z", "2016-01-01T00:00:00Z" ]`
> - `[ "2015-01-01", "2016-01-01" ]`

`dimension = null`

The name of the temporal dimension to filter on. If no specific dimension is specified or it is set to `null`, the filter applies to all temporal dimensions. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

> Data type: **string, null**

Return Value

A data cube restricted to the specified temporal extent. The dimensions and dimension properties (name, type, labels, reference system and resolution) remain unchanged, except that the temporal dimensions (determined by `dimensions` parameter) may have less dimension labels.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

See Also

- Filters explained in the openEO documentation

first

First element

ARRAYS REDUCER

Download JSON

Description

```
first(array data, ?boolean ignore_nodata = true) : any
```

Gives the first element of an array.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array with elements of any data type.

> Data type: **array**
>
> Array items:
>
> > Any data type is allowed.
> >
> > Data type: **any**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if the first value is such a value.

> Data type: **boolean**

Return Value

The first element of the input array.

> Any data type is allowed.
>
> Data type: **any**

Examples

Example #1

```
first(data = [1,0,3,2]) => 1
```

Example #2

```
first(data = [null,"A","B"]) => "A"
```

Example #3

```
first(data = [null,2,3], ignore_nodata = false) => null
```

Example #4

The input array is empty: return `null`.

```
first(data = []) => null
```

fit_curve

Curve fitting — **experimental**

CUBES MATH

Download JSON

Description

```
fit_curve(raster-cube data, array<number>|raster-cube parameters, process-graph:object
function, string dimension) : raster-cube
```

Use non-linear least squares to fit a model function `y = f(x, parameters)` to data.

The process throws an `InvalidValues` exception if invalid values are encountered. Invalid values are finite numbers (see also `is_valid`).

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

data*

A data cube.

Data type: **raster-cube**

parameters*

Defined the number of parameters for the model function and provides an initial guess for them. At least one parameter is required.

Data Types:

Data type: **array<number>**

Min. number of items: 1

Array items:

Data type: **number**

Data Cube with optimal values from a previous result of this process.

Data type: **raster-cube**

function*

The model function. It must take the parameters to fit as array through the first argument and the independent variable `x` as the second argument.

It is recommended to store the model function as a user-defined process on the back-end to be able to re-use the model function with the computed optimal values for the parameters afterwards.

Data type: **User-defined Process (process-graph:object)**

Parameters:

x*

The value for the independent variable `x`.

Data type: **number**

parameters*

The parameters for the model function, contains at least one parameter.

Data type: **array<number>**

Min. number of items: 1

Array items: Data type: **number**

Expected Return Value:

The computed value `y` value for the given independent variable `x` and the parameters.

Data type: **number**

`dimension`*

The name of the dimension for curve fitting. Must be a dimension with labels that have a order (i.e. numerical labels or a temporal dimension). Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

Data type: **string**

Return Value

A data cube with the optimal values for the parameters.

Data type: **raster-cube**

Errors/Exceptions

- `InvalidValues`
  Message: *At least one of the values is not a finite number.*
- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

floor

Round fractions down

MATH > ROUNDING

Download JSON

Description

`floor(number|null x) : integer|null`

The greatest integer less than or equal to the number `x`.

This process is *not* an alias for the `int` process as defined by some mathematicians, see the examples for negative numbers in both processes for differences.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number to round down.

> Data type: **number, null**

Return Value

The number rounded down.

> Data type: **integer, null**

Examples

Example #1

```
floor(x = 0) => 0
```

Example #2

```
floor(x = 3.5) => 3
```

Example #3

```
floor(x = -0.4) => -1
```

Example #4

```
floor(x = -3.5) => -4
```

See Also

- Floor explained by Wolfram MathWorld

gt

Greater than comparison

COMPARISON

Download JSON

Description

```
gt(any x, any y) : boolean|null
```

Compares whether `x` is strictly greater than `y`.

Remarks:

- If any operand is `null`, the return value is `null`.
- If any operand is an array or object, the return value is `false`.
- If any operand is not a `number` or temporal string (`date`, `time` or `date-time`), the process returns `false`.
- Temporal strings can *not* be compared based on their string representation due to the time zone / time-offset representations.

Parameters

x*

First operand.

> Any data type is allowed.
>
> Data type: **any**

y*

Second operand.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if `x` is strictly greater than `y` or `null` if any operand is `null`, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
gt(x = 1, y = null) => null
```

Example #2

```
gt(x = 0, y = 0) => false
```

Example #3

```
gt(x = 2, y = 1) => true
```

Example #4

```
gt(x = -0.5, y = -0.6) => true
```

Example #5

```
gt(x = "00:00:00Z", y = "00:00:00+01:00") => true
```

Example #6

```
gt(x = "1950-01-01T00:00:00Z", y = "2018-01-01T12:00:00Z") => false
```

Example #7

```
gt(x = "2018-01-01T12:00:00+00:00", y = "2018-01-01T12:00:00Z") => false
```

Example #8

```
gt(x = true, y = 0) => false
```

Example #9

```
gt(x = true, y = false) => false
```

gte

Greater than or equal to comparison

COMPARISON

Download JSON

Description

`gte(any x, any y) : boolean|null`

Compares whether `x` is greater than or equal to `y`.

Remarks:

- If any operand is `null`, the return value is `null`. Therefore, `gte(null, null)` returns `null` instead of `true`.
- If any operand is an array or object, the return value is `false`.
- If the operands are not equal (see process `eq`) and any of them is not a `number` or temporal string (`date`, `time` or `date-time`), the process returns `false`.
- Temporal strings can *not* be compared based on their string representation due to the time zone / time-offset representations.

Parameters

x*

First operand.

> Any data type is allowed.
>
> Data type: **any**

y*

Second operand.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if `x` is greater than or equal to `y`, `null` if any operand is `null`, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
gte(x = 1, y = null) => null
```

Example #2

```
gte(x = 0, y = 0) => true
```

Example #3

```
gte(x = 1, y = 2) => false
```

Example #4

```
gte(x = -0.5, y = -0.6) => true
```

Example #5

```
gte(x = "00:00:00Z", y = "00:00:00+01:00") => true
```

Example #6

```
gte(x = "1950-01-01T00:00:00Z", y = "2018-01-01T12:00:00Z") => false
```

Example #7

```
gte(x = "2018-01-01T12:00:00+00:00", y = "2018-01-01T12:00:00Z") => true
```

Example #8

```
gte(x = true, y = false) => false
```

Example #9

```
gte(x = [1,2,3], y = [1,2,3]) => false
```

if

If-Then-Else conditional

LOGIC COMPARISON MASKS

Download JSON

Description

```
if(boolean|null value, any accept, ?any reject = null) : any
```

If the value passed is `true`, returns the value of the `accept` parameter, otherwise returns the value of the `reject` parameter.

This is basically an if-then-else construct as in other programming languages.

Parameters

`value`*

A boolean value.

> Data type: **boolean, null**

`accept`*

A value that is returned if the boolean value is `true`.

> Any data type is allowed.
>
> Data type: **any**

`reject = null`

A value that is returned if the boolean value is **not** `true`. Defaults to `null`.

> Any data type is allowed.
>
> Data type: **any**

Return Value

Either the `accept` or `reject` argument depending on the given boolean value.

> Any data type is allowed.
>
> Data type: **any**

Examples

Example #1

```
if(value = true, accept = "A", reject = "B") => "A"
```

Example #2

```
if(value = null, accept = "A", reject = "B") => "B"
```

Example #3

```
if(value = false, accept = [1,2,3], reject = [4,5,6]) => [4,5,6]
```

Example #4

```
if(value = true, accept = 123) => 123
```

Example #5

```
if(value = false, accept = 1) => null
```

inspect

Add information to the logs — **experimental**

DEVELOPMENT

Download JSON

Description

```
inspect(any data, ?string code = "User", ?string level = "info", ?string message = "") : any
```

This process can be used to add runtime information to the logs, e.g. for debugging purposes. This process should be used with caution and it is recommended to remove the process in production workflows. For example, logging each pixel or array individually in a process such as `apply` or `reduce_dimension` could lead to a (too) large number of log entries. Several data structures (e.g. data cubes) are too large to log and will only return summaries of their contents.

The data provided in the parameter `data` is returned without changes.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

Data to log.

> Any data type is allowed.
>
> Data type: **any**

`code = "User"`

A label to help identify one or more log entries originating from this process in the list of all log entries. It can help to group or filter log entries and is usually not unique.

> Data type: **string**

`level = "info"`

The severity level of this message, defaults to `info`.

> Data type: **string**
>
> Allowed values: error, warning, info, debug

`message = ""`

A message to send in addition to the data.

> Data type: **string**

Return Value

The data as passed to the `data` parameter without any modification.

> Any data type is allowed.
>
> Data type: **any**

int

Integer part of a number

MATH MATH > ROUNDING

Download JSON

Description

`int(number|null x) : integer|null`

The integer part of the real number `x`.

This process is *not* an alias for the `floor` process as defined by some mathematicians, see the examples for negative numbers in both processes for differences.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

Integer part of the number.

> Data type: **integer, null**

Examples

Example #1

```
int(x = 0) => 0
```

Example #2

```
int(x = 3.5) => 3
```

Example #3

```
int(x = -0.4) => 0
```

Example #4

```
int(x = -3.5) => -3
```

See Also

- Integer Part explained by Wolfram MathWorld

is_infinite

Value is an infinite number — **experimental**

COMPARISON

Download JSON

Description

`is_infinite(any x) : boolean`

Checks whether the specified value `x` is an infinite number. The definition of infinite numbers follows the IEEE Standard 754. The special numerical value `NaN` (not a number) as defined by the IEEE Standard 754 is not an infinite number and must return `false`.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

x*

The data to check.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if the data is an infinite number, otherwise `false`.

> Data type: **boolean**

See Also

- IEEE Standard 754-2008 for Floating-Point Arithmetic

is_nan

Value is not a number

COMPARISON MATH > CONSTANTS

Download JSON

Description

`is_nan(any x) : boolean`

Checks whether the specified value `x` is not a number. Returns `true` for numeric values (integers and floating-point numbers), except for the special value `NaN` as defined by the IEEE Standard 754. All non-numeric data types MUST also return `true`, including arrays that contain `NaN` values.

Parameters

x*

The data to check.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if the data is not a number, otherwise `false`.

> Data type: **boolean**

Examples

Example #1

```
is_nan(x = 1) => false
```

Example #2

```
is_nan(x = "Test") => true
```

Example #3

```
is_nan(x = null) => true
```

See Also

- IEEE Standard 754-2008 for Floating-Point Arithmetic
- NaN explained by Wolfram MathWorld

is_nodata

Value is a no-data value

COMPARISON

Download JSON

Description

`is_nodata(any x) : boolean`

Checks whether the specified data is missing data, i.e. equals to `null` or any of the no-data values specified in the metadata. The special numerical value `NaN` (not a number) as defined by the IEEE Standard 754 is not considered no-data and must return `false`.

Parameters

x*

The data to check.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if the data is a no-data value, otherwise `false`.

> Data type: **boolean**

Examples

Example #1

```
is_nodata(x = 1) => false
```

Example #2

```
is_nodata(x = "Test") => false
```

Example #3

```
is_nodata(x = null) => true
```

Example #4

```
is_nodata(x = [null,null]) => false
```

is_valid

Value is valid data

COMPARISON

Download JSON

Description

`is_valid(any x) : boolean`

Checks whether the specified value `x` is valid. The following values are considered valid:

- Any finite numerical value (integers and floating-point numbers). The definition of finite numbers follows the IEEE Standard 754 and excludes the special value `NaN` (not a number).
- Any other value that is not a no-data value according to ``is_nodata()`. Thus all arrays, objects and strings are valid, regardless of their content.

Parameters

x*

The data to check.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if the data is valid, otherwise `false`.

> Data type: **boolean**

Examples

Example #1

```
is_valid(x = 1) => true
```

Example #2

```
is_valid(x = "Test") => true
```

Example #3

```
is_valid(x = null) => false
```

Example #4

```
is_valid(x = [null,null]) => true
```

See Also

- IEEE Standard 754-2008 for Floating-Point Arithmetic

last

Last element

ARRAYS REDUCER

Download JSON

Description

`last(array data, ?boolean ignore_nodata = true) : any`

Gives the last element of an array.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array with elements of any data type.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if the last value is such a value.

Data type: **boolean**

Return Value

The last element of the input array.

Any data type is allowed.

> Data type: **any**

Examples

Example #1

```
last(data = [1,0,3,2]) => 2
```

Example #2

```
last(data = ["A","B",null]) => "B"
```

Example #3

```
last(data = [0,1,null], ignore_nodata = false) => null
```

Example #4

> The input array is empty: return `null`.

```
last(data = []) => null
```

linear_scale_range

Linear transformation between two ranges

MATH

Download JSON

Description

```
linear_scale_range(number|null x, number inputMin, number inputMax, ?number outputMin = 0, ?number outputMax = 1) : number|null
```

Performs a linear transformation between the input and output range.

The given number in `x` is clipped to the bounds specified in `inputMin` and `inputMax` so that the underlying formula *`((x - inputMin) / (inputMax - inputMin)) * (outputMax - outputMin) + outputMin`* never returns any value lower than `outputMin` or greater than `outputMax`.

Potential use case include

- scaling values to the 8-bit range (0 - 255) often used for numeric representation of values in one of the channels of the RGB colour model or
- calculating percentages (0 - 100).

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number to transform. The number gets clipped to the bounds specified in `inputMin` and `inputMax`.

> Data type: **number, null**

inputMin*

Minimum value the input can obtain.

> Data type: **number**

inputMax*

Maximum value the input can obtain.

> Data type: **number**

outputMin = 0

Minimum value of the desired output range.

> Data type: **number**

outputMax = 1

Maximum value of the desired output range.

> Data type: **number**

Return Value

The transformed number.

> Data type: **number, null**

Examples

Example #1

```
linear_scale_range(x = 0.3, inputMin = -1, inputMax = 1, outputMin = 0, outputMax = 255) => 165.75
```

Example #2

```
linear_scale_range(x = 25.5, inputMin = 0, inputMax = 255) => 0.1
```

Example #3

```
linear_scale_range(x = null, inputMin = 0, inputMax = 100) => null
```

Example #4

Shows that the input data is clipped.

```
linear_scale_range(x = 1.12, inputMin = 0, inputMax = 1, outputMin = 0, outputMax = 255) => 255
```

ln

Natural logarithm

MATH > EXPONENTIAL & LOGARITHMIC

Download JSON

Description

`ln(number|null x) : number|null`

The natural logarithm is the logarithm to the base *e* of the number `x`, which equals to using the *log* process with the base set to *e*. The natural logarithm is the inverse function of taking *e* to the power x.

The no-data value `null` is passed through.

The computations follow IEEE Standard 754 whenever the processing environment supports it. Therefore, *`ln(0)`* results in ±infinity if the processing environment supports it or otherwise an exception is thrown.

Parameters

x*

A number to compute the natural logarithm for.

Data type: **number, null**

Return Value

The computed natural logarithm.

Data type: **number, null**

Examples

Example #1

```
ln(x = 1) => 0
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Natural logarithm explained by Wolfram MathWorld

load_collection

Load a collection

CUBES IMPORT

Download JSON

Description

```
load_collection(collection-id:string id, object|null spatial_extent, temporal-
interval:array<string|null>|null temporal_extent, ?array<band-name:string>|null bands = null,
?metadata-filter:object|null properties = null) : raster-cube
```

Loads a collection from the current back-end by its id and returns it as a processable data cube. The data that is added to the data cube can be restricted with the parameters `spatial_extent`, `temporal_extent`, `bands` and `properties`.

Remarks:

- The bands (and all dimensions that specify nominal dimension labels) are expected to be ordered as specified in the metadata if the `bands` parameter is set to `null`.
- If no additional parameter is specified this would imply that the whole data set is expected to be loaded. Due to the large size of many data sets, this is not recommended and may be optimized by back-ends to only load the data that is actually required after evaluating subsequent processes such as filters. This means that the pixel values should be processed only after the data has been limited to the required extent and as a consequence also to a manageable size.

Parameters

id*

The collection id.

> Data type: **collection-id:string**
>
> Pattern: `^[\w\-\.~/]+$`

spatial_extent*

Limits the data to load from the collection to the specified bounding box or polygons.

The process puts a pixel into the data cube if the point at the pixel center intersects with the bounding box or any of the polygons (as defined in the Simple Features standard by the OGC).

The GeoJSON can be one of the following feature types:

- A `Polygon` or `MultiPolygon` geometry,
- a `Feature` with a `Polygon` or `MultiPolygon` geometry,
- a `FeatureCollection` containing at least one `Feature` with `Polygon` or `MultiPolygon` geometries, or
- a `GeometryCollection` containing `Polygon` or `MultiPolygon` geometries. To maximize interoperability, `GeometryCollection` should be avoided in favour of one of the alternatives above.

Set this parameter to `null` to set no limit for the spatial extent. Be careful with this when loading large datasets! It is recommended to use this parameter instead of using `filter_bbox` or `filter_spatial` directly after loading

unbounded data.

Data Types:

Bounding Box

Data type: **bounding-box:object**

Object Properties:

west * — West (lower left corner, coordinate axis 1).

Data type: **number**

south * — South (lower left corner, coordinate axis 2).

Data type: **number**

east * — East (upper right corner, coordinate axis 1).

Data type: **number**

north * — North (upper right corner, coordinate axis 2).

Data type: **number**

base — Base (optional, lower left corner, coordinate axis 3).

Data type: **number, null**

Default value: `null`

height — Height (optional, upper right corner, coordinate axis 3).

Data type: **number, null**

Default value: `null`

crs — Coordinate reference system of the extent, specified as as EPSG code, WKT2 (ISO 19162) string or PROJ definition (deprecated). Defaults to `4326` (EPSG code 4326) unless the client explicitly requests a different coordinate reference system.

Data type: **any**

Default value: 4326

GeoJSON

Limits the data cube to the bounding box of the given geometry. All pixels inside the bounding box that do not intersect with any of the polygons will be set to no data (`null`).

Data type: **geojson:object**

No filter

Don't filter spatially. All data is included in the data cube.

Data type: **null**

temporal_extent*

Limits the data to load from the collection to the specified left-closed temporal interval. Applies to all temporal dimensions. The interval has to be specified as an array with exactly two elements:

1. The first element is the start of the temporal interval. The specified instance in time is **included** in the interval.
2. The second element is the end of the temporal interval. The specified instance in time is **excluded** from the interval.

The specified temporal strings follow RFC 3339. Also supports open intervals by setting one of the boundaries to `null`, but never both.

Set this parameter to `null` to set no limit for the temporal extent. Be careful with this when loading large datasets! It is recommended to use this parameter instead of using `filter_temporal` directly after loading unbounded data.

Data Types:

Data type: **temporal-interval:array<date-time:string|date:string|year:string|null>**

Min. number of items: 2

Max. number of items: 2

Array items: Data type: **any**

Examples:

- `[ "2015-01-01T00:00:00Z", "2016-01-01T00:00:00Z" ]`
- `[ "2015-01-01", "2016-01-01" ]`

No filter

Don't filter temporally. All data is included in the data cube.

Data type: **null**

bands = null

Only adds the specified bands into the data cube so that bands that don't match the list of band names are not available. Applies to all dimensions of type `bands`.

Either the unique band name (metadata field `name` in bands) or one of the common band names (metadata field `common_name` in bands) can be specified. If the unique band name and the common name conflict, the unique band name has a higher priority.

The order of the specified array defines the order of the bands in the data cube. If multiple bands match a common name, all matched bands are included in the original order.

It is recommended to use this parameter instead of using `filter_bands` directly after loading unbounded data.

Data Types:

Data type: **array<band-name:string>**

Array items: Data type: **band-name:string**

No filter

Don't filter bands. All bands are included in the data cube.

Data type: **null**

`properties` = null

Limits the data by metadata properties to include only data in the data cube which all given conditions return `true` for (AND operation).

Specify key-value-pairs with the key being the name of the metadata property, which can be retrieved with the openEO Data Discovery for Collections. The value must be a condition (user-defined process) to be evaluated against the collection metadata, see the example.

Data Types:

Filters

A list of filters to check against. Specify key-value-pairs with the key being the name of the metadata property name and the value being a process evaluated against the metadata values.

Data type: **metadata-filter:object**

Each property:

Data type: **User-defined Process (process-graph:object)**

Parameters:

`value`*

The property value to be checked against.

Any data type.

Data type: **any**

Expected Return Value:

`true` if the data should be loaded into the data cube, otherwise `false`.

Data type: **boolean**

No filter

Don't filter by metadata properties.

Data type: **null**

Return Value

A data cube for further processing. The dimensions and dimension properties (name, type, labels, reference system and resolution) correspond to the collection's metadata, but the dimension labels are restricted as specified in the parameters.

Data type: **raster-cube**

Examples

Example #1

Loading `Sentinel-2B` data from a `Sentinel-2` collection for 2018, but only with cloud cover between 0 and 50%.

```
load_collection(id = "Sentinel-2", spatial_extent =
{"west":16.1,"east":16.6,"north":48.6,"south":47.2}, temporal_extent = ["2018-01-
01","2019-01-01"], properties = {"eo:cloud_cover":{"process_graph":{"cc":
{"process_id":"between","arguments":{"x":
{"from_parameter":"value"},"min":0,"max":50},"result":true}}},"platform":
{"process_graph":{"pf":{"process_id":"eq","arguments":{"x":
{"from_parameter":"value"},"y":"Sentinel-
2B","case_sensitive":false},"result":true}}}})
```

See Also

- Data Cubes explained in the openEO documentation
- List of common band names as specified by the STAC specification
- Official EPSG code registry
- PROJ parameters for cartographic projections
- Simple Features standard by the OGC
- Unofficial EPSG code database

load_result

Load batch job results — **experimental**

CUBES IMPORT

Download JSON

Description

```
load_result(string id, ?object|null spatial_extent = null, ?temporal-
interval:array<string|null>|null temporal_extent = null, ?array<band-name:string>|null bands
= null) : raster-cube
```

Loads batch job results and returns them as a processable data cube. A batch job result can be loaded by ID or URL:

- **ID**: The identifier for a finished batch job. The job must have been submitted by the authenticated user on the back-end currently connected to.
- **URL**: The URL to the STAC metadata for a batch job result. This is usually a signed URL that is provided by some back-ends since openEO API version 1.1.0 through the `canonical` link relation in the batch job result metadata.

If supported by the underlying metadata and file format, the data that is added to the data cube can be restricted with the parameters `spatial_extent`, `temporal_extent` and `bands`.

Remarks:

- The bands (and all dimensions that specify nominal dimension labels) are expected to be ordered as specified in the metadata if the `bands` parameter is set to `null`.
- If no additional parameter is specified this would imply that the whole data set is expected to be loaded. Due to the large size of many data sets, this is not recommended and may be optimized by back-ends to only load the data that is actually required after evaluating subsequent processes such as filters. This means that the pixel values should be processed only after the data has been limited to the required extent and as a consequence also to a manageable size.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`id`*

The id of a batch job with results.

Data Types:

> **ID**
>
> Data type: **job-id:string**
>
> Pattern: `^[\w\-\.~]+$`

> **URL**
>
> Data type: **uri:string**
>
> Pattern: `^https?://`

`spatial_extent` = null

Limits the data to load from the batch job result to the specified bounding box or polygons.

The process puts a pixel into the data cube if the point at the pixel center intersects with the bounding box or any of the polygons (as defined in the Simple Features standard by the OGC).

The GeoJSON can be one of the following feature types:

- A `Polygon` or `MultiPolygon` geometry,
- a `Feature` with a `Polygon` or `MultiPolygon` geometry,
- a `FeatureCollection` containing at least one `Feature` with `Polygon` or `MultiPolygon` geometries, or
- a `GeometryCollection` containing `Polygon` or `MultiPolygon` geometries. To maximize interoperability, `GeometryCollection` should be avoided in favour of one of the alternatives above.

Set this parameter to `null` to set no limit for the spatial extent. Be careful with this when loading large datasets! It is recommended to use this parameter instead of using `filter_bbox` or `filter_spatial` directly after loading unbounded data.

Data Types:

Bounding Box

Data type: **bounding-box:object**

Object Properties:

west * West (lower left corner, coordinate axis 1).

Data type: **number**

south * South (lower left corner, coordinate axis 2).

Data type: **number**

east * East (upper right corner, coordinate axis 1).

Data type: **number**

north * North (upper right corner, coordinate axis 2).

Data type: **number**

base Base (optional, lower left corner, coordinate axis 3).

Data type: **number, null**

Default value: `null`

height Height (optional, upper right corner, coordinate axis 3).

Data type: **number, null**

Default value: `null`

crs Coordinate reference system of the extent, specified as as EPSG code, WKT2 (ISO 19162) string or PROJ definition (deprecated). Defaults to `4326` (EPSG code 4326) unless the client explicitly requests a different coordinate reference system.

Data type: **any**

Default value: 4326

GeoJSON

Limits the data cube to the bounding box of the given geometry. All pixels inside the bounding box that do not intersect with any of the polygons will be set to no data (`null`).

Data type: **geojson:object**

No filter

Don't filter spatially. All data is included in the data cube.

Data type: **null**

`temporal_extent` = `null`

Limits the data to load from the batch job result to the specified left-closed temporal interval. Applies to all temporal dimensions. The interval has to be specified as an array with exactly two elements:

1. The first element is the start of the temporal interval. The specified instance in time is **included** in the interval.
2. The second element is the end of the temporal interval. The specified instance in time is **excluded** from the interval.

The specified temporal strings follow RFC 3339. Also supports open intervals by setting one of the boundaries to `null`, but never both.

Set this parameter to `null` to set no limit for the temporal extent. Be careful with this when loading large datasets! It is recommended to use this parameter instead of using `filter_temporal` directly after loading unbounded data.

Data Types:

Data type: **temporal-interval:array<date-time:string|date:string|year:string|null>**

Min. number of items: 2

Max. number of items: 2

Array items: Data type: **any**

Examples:

- `[ "2015-01-01T00:00:00Z", "2016-01-01T00:00:00Z" ]`
- `[ "2015-01-01", "2016-01-01" ]`

No filter

Don't filter temporally. All data is included in the data cube.

Data type: **null**

`bands = null`

Only adds the specified bands into the data cube so that bands that don't match the list of band names are not available. Applies to all dimensions of type `bands`.

Either the unique band name (metadata field `name` in bands) or one of the common band names (metadata field `common_name` in bands) can be specified. If the unique band name and the common name conflict, the unique band name has a higher priority.

The order of the specified array defines the order of the bands in the data cube. If multiple bands match a common name, all matched bands are included in the original order.

It is recommended to use this parameter instead of using `filter_bands` directly after loading unbounded data.

Data Types:

Data type: **array<band-name:string>**

Array items: Data type: **band-name:string**

No filter

Don't filter bands. All bands are included in the data cube.

Data type: **null**

Return Value

A data cube for further processing.

Data type: **raster-cube**

load_uploaded_files

Load files from the user workspace — **experimental**

CUBES IMPORT

Download JSON

Description

```
load_uploaded_files(file-paths:array<file-path:string> paths, input-format:string format, ?
input-format-options:object options = {}) : raster-cube
```

Loads one or more user-uploaded files from the server-side workspace of the authenticated user and returns them as a single data cube. The files must have been stored by the authenticated user on the back-end currently connected to.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`paths`*

The files to read. Folders can't be specified, specify all files instead. An exception is thrown if a file can't be read.

Data type: **file-paths:array<file-path:string>**

Array items:

Data type: **file-path:string**

Pattern: `^[^\r\n\:'"]+$`

`format`*

The file format to read from. It must be one of the values that the server reports as supported input file formats, which usually correspond to the short GDAL/OGR codes. If the format is not suitable for loading the data, a `FormatUnsuitable` exception will be thrown. This parameter is *case insensitive*.

Data type: **input-format:string**

`options = {}`

The file format parameters to be used to read the files. Must correspond to the parameters that the server reports as supported parameters for the chosen `format`. The parameter names and valid values usually correspond to the GDAL/OGR format options.

Data type: **input-format-options:object**

Return Value

A data cube for further processing.

Data type: **raster-cube**

Errors/Exceptions

- `FormatUnsuitable`

  Message: *Data can't be loaded with the requested input format.*

log

Logarithm to a base

MATH > EXPONENTIAL & LOGARITHMIC

Download JSON

Description

`log(number|null x, number|null base) : number|null`

Logarithm to the base `base` of the number `x` is defined to be the inverse function of taking b to the power of x.

The no-data value `null` is passed through and therefore gets propagated if any of the arguments is `null`.

The computations follow IEEE Standard 754 whenever the processing environment supports it. Therefore, `log(0, 2)` results in ±infinity if the processing environment supports it or otherwise an exception is thrown.

Parameters

x*

A number to compute the logarithm for.

> Data type: **number, null**

base*

The numerical base.

> Data type: **number, null**

Return Value

The computed logarithm.

> Data type: **number, null**

Examples

Example #1

```
log(x = 10, base = 10) => 1
```

Example #2

```
log(x = 2, base = 2) => 1
```

Example #3

```
log(x = 4, base = 2) => 2
```

Example #4

```
log(x = 1, base = 16) => 0
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Logarithm explained by Wolfram MathWorld

lt

Less than comparison

COMPARISON

Download JSON

Description

`lt(any x, any y) : boolean|null`

Compares whether `x` is strictly less than `y`.

Remarks:

- If any operand is `null`, the return value is `null`.
- If any operand is an array or object, the return value is `false`.
- If any operand is not a `number` or temporal string (`date`, `time` or `date-time`), the process returns `false`.
- Temporal strings can *not* be compared based on their string representation due to the time zone / time-offset representations.

Parameters

x*

First operand.

> Any data type is allowed.
>
> Data type: **any**

y*

Second operand.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if `x` is strictly less than `y`, `null` if any operand is `null`, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
lt(x = 1, y = null) => null
```

Example #2

```
lt(x = 0, y = 0) => false
```

Example #3

```
lt(x = 1, y = 2) => true
```

Example #4

```
lt(x = -0.5, y = -0.6) => false
```

Example #5

```
lt(x = "00:00:00+01:00", y = "00:00:00Z") => true
```

Example #6

```
lt(x = "1950-01-01T00:00:00Z", y = "2018-01-01T12:00:00Z") => true
```

Example #7

```
lt(x = "2018-01-01T12:00:00+00:00", y = "2018-01-01T12:00:00Z") => false
```

Example #8

```
lt(x = 0, y = true) => false
```

Example #9

```
lt(x = false, y = true) => false
```

lte

Less than or equal to comparison

Download JSON

Description

`lte(any x, any y) : boolean|null`

Compares whether `x` is less than or equal to `y`.

Remarks:

- If any operand is `null`, the return value is `null`. Therefore, `lte(null, null)` returns `null` instead of `true`.
- If any operand is an array or object, the return value is `false`.
- If the operands are not equal (see process `eq`) and any of them is not a `number` or temporal string (`date`, `time` or `date-time`), the process returns `false`.
- Temporal strings can *not* be compared based on their string representation due to the time zone / time-offset representations.

Parameters

x*

First operand.

> Any data type is allowed.
>
> Data type: **any**

y*

Second operand.

> Any data type is allowed.
>
> Data type: **any**

Return Value

`true` if `x` is less than or equal to `y`, `null` if any operand is `null`, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
lte(x = 1, y = null) => null
```

Example #2

```
lte(x = 0, y = 0) => true
```

Example #3

```
lte(x = 1, y = 2) => true
```

Example #4

```
lte(x = -0.5, y = -0.6) => false
```

Example #5

```
lte(x = "00:00:00+01:00", y = "00:00:00Z") => true
```

Example #6

```
lte(x = "1950-01-01T00:00:00Z", y = "2018-01-01T12:00:00Z") => true
```

Example #7

```
lte(x = "2018-01-01T12:00:00+00:00", y = "2018-01-01T12:00:00Z") => true
```

Example #8

```
lte(x = false, y = true) => false
```

Example #9

```
lte(x = [1,2,3], y = [1,2,3]) => false
```

mask

Apply a raster mask

CUBES MASKS

Download JSON

Description

```
mask(raster-cube data, raster-cube mask, ?number|boolean|string|null replacement = null) :
raster-cube
```

Applies a mask to a raster data cube. To apply a vector mask use `mask_polygon`.

A mask is a raster data cube for which corresponding pixels among `data` and `mask` are compared and those pixels in `data` are replaced whose pixels in `mask` are non-zero (for numbers) or `true` (for boolean values). The pixel values are replaced with the value specified for `replacement`, which defaults to `null` (no data).

The data cubes have to be compatible so that each dimension in the mask must also be available in the raster data cube with the same name, type, reference system, resolution and labels. Dimensions can be missing in the mask with the result that the

mask is applied for each label of the missing dimension in the data cube. The process fails if there's an incompatibility found between the raster data cube and the mask.

Parameters

`data`*

A raster data cube.

> Data type: **raster-cube**

`mask`*

A mask as a raster data cube. Every pixel in `data` must have a corresponding element in `mask`.

> Data type: **raster-cube**

`replacement` = `null`

The value used to replace masked values with.

> Data type: **number, boolean, string, null**

Return Value

A masked raster data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

> Data type: **raster-cube**

mask_polygon

Apply a polygon mask

CUBES MASKS

Download JSON

Description

```
mask_polygon(raster-cube data, geojson:object mask, ?number|boolean|string|null replacement =
null, ?boolean inside = false) : raster-cube
```

Applies a (multi) polygon mask to a raster data cube. To apply a raster mask use `mask`.

All pixels for which the point at the pixel center **does not** intersect with any polygon (as defined in the Simple Features standard by the OGC) are replaced. This behavior can be inverted by setting the parameter `inside` to `true`.

The pixel values are replaced with the value specified for `replacement`, which defaults to `null` (no data). No data values in `data` will be left untouched by the masking operation.

Parameters

`data`*

A raster data cube.

> Data type: **raster-cube**

`mask`*

A GeoJSON object containing at least one polygon. The provided feature types can be one of the following:

- A `Polygon` or `MultiPolygon` geometry,
- a `Feature` with a `Polygon` or `MultiPolygon` geometry,
- a `FeatureCollection` containing at least one `Feature` with `Polygon` or `MultiPolygon` geometries, or
- a `GeometryCollection` containing `Polygon` or `MultiPolygon` geometries. To maximize interoperability, `GeometryCollection` should be avoided in favour of one of the alternatives above.

> Data type: **geojson:object**

`replacement = null`

The value used to replace masked values with.

Data Types:

> Data type: **number**

> Data type: **boolean**

> Data type: **string**

> Data type: **null**

`inside = false`

If set to `true` all pixels for which the point at the pixel center **does** intersect with any polygon are replaced.

> Data type: **boolean**

Return Value

A masked raster data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

> Data type: **raster-cube**

See Also

- Simple Features standard by the OGC

max

Maximum value

MATH MATH > STATISTICS REDUCER Download JSON

Description

```
max(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

Computes the largest value of an array of numbers, which is equal to the first element of a sorted (i.e., ordered) version of the array.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The maximum value.

> Data type: **number, null**

Examples

Example #1

```
max(data = [1,0,3,2]) => 3
```

Example #2

```
max(data = [5,2.5,null,-0.7]) => 5
```

Example #3

```
max(data = [1,0,3,null,2], ignore_nodata = false) => null
```

Example #4

The input array is empty: return `null`.

```
max(data = []) => null
```

See Also

- Maximum explained by Wolfram MathWorld

mean

Arithmetic mean (average)

MATH > STATISTICS REDUCER

Download JSON

Description

```
mean(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

The arithmetic mean of an array of numbers is the quantity commonly called the average. It is defined as the sum of all elements divided by the number of elements.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The computed arithmetic mean.

> Data type: **number, null**

Examples

Example #1

```
mean(data = [1,0,3,2]) => 1.5
```

Example #2

```
mean(data = [9,2.5,null,-2.5]) => 3
```

Example #3

```
mean(data = [1,null], ignore_nodata = false) => null
```

Example #4

> The input array is empty: return `null`.

```
mean(data = []) => null
```

Example #5

> The input array has only `null` elements: return `null`.

```
mean(data = [null,null]) => null
```

See Also

- Arithmetic mean explained by Wolfram MathWorld

median

Statistical median

MATH > STATISTICS REDUCER Download JSON

Description

```
median(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

The statistical median of an array of numbers is the value separating the higher half from the lower half of the data.

An array without non-`null` elements resolves always with `null`.

Remarks:

- For symmetric arrays, the result is equal to the `mean`.
- The median can also be calculated by computing the `quantiles` with a probability of *0.5*.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The computed statistical median.

> Data type: **number, null**

Examples

Example #1

```
median(data = [1,3,3,6,7,8,9]) => 6
```

Example #2

```
median(data = [1,2,3,4,5,6,8,9]) => 4.5
```

Example #3

```
median(data = [-1,-0.5,null,1]) => -0.5
```

Example #4

```
median(data = [-1,0,null,1], ignore_nodata = false) => null
```

Example #5

> The input array is empty: return `null`.

```
median(data = []) => null
```

Example #6

> The input array has only `null` elements: return `null`.
>
> ```
> median(data = [null,null]) => null
> ```

See Also

- Statistical Median explained by Wolfram MathWorld

merge_cubes

Merge two data cubes

CUBES

Download JSON

Description

```
merge_cubes(raster-cube cube1, raster-cube cube2, ?process-graph:object overlap_resolver =
null, ?any context = null) : raster-cube
```

The data cubes have to be compatible. A merge operation without overlap should be reversible with (a set of) filter operations for each of the two cubes. The process performs the join on overlapping dimensions, with the same name and type.

An overlapping dimension has the same name, type, reference system and resolution in both dimensions, but can have different labels. One of the dimensions can have different labels, for all other dimensions the labels must be equal. If data overlaps, the parameter `overlap_resolver` must be specified to resolve the overlap.

Examples for merging two data cubes:

1. Data cubes with the dimensions (`x`, `y`, `t`, `bands`) have the same dimension labels in `x`, `y` and `t`, but the labels for the dimension `bands` are `B1` and `B2` for the first cube and `B3` and `B4`. An overlap resolver is *not needed*. The merged data cube has the dimensions `x`, `y`, `t` and `bands` and the dimension `bands` has four dimension labels: `B1`, `B2`, `B3`, `B4`.
2. Data cubes with the dimensions (`x`, `y`, `t`, `bands`) have the same dimension labels in `x`, `y` and `t`, but the labels for the dimension `bands` are `B1` and `B2` for the first data cube and `B2` and `B3` for the second. An overlap resolver is *required* to resolve overlap in band `B2`. The merged data cube has the dimensions `x`, `y`, `t` and `bands` and the dimension `bands` has three dimension labels: `B1`, `B2`, `B3`.
3. Data cubes with the dimensions (`x`, `y`, `t`) have the same dimension labels in `x`, `y` and `t`. There are two options:
    1. Keep the overlapping values separately in the merged data cube: An overlap resolver is *not needed*, but for each data cube you need to add a new dimension using `add_dimension`. The new dimensions must be equal, except that the labels for the new dimensions must differ by name. The merged data cube has the same dimensions and labels as the original data cubes, plus the dimension added with `add_dimension`, which has the two dimension labels after the merge.
    2. Combine the overlapping values into a single value: An overlap resolver is *required* to resolve the overlap for all pixels. The merged data cube has the same dimensions and labels as the original data cubes, but all pixel values have been processed by the overlap resolver.
4. A data cube with dimensions (`x`, `y`, `t` / `bands`) or (`x`, `y`, `t`, `bands`) and another data cube with dimensions (`x`, `y`) have the same dimension labels in `x` and `y`. Merging them will join dimensions `x` and `y`, so the lower dimension cube is merged with each time step and band available in the higher dimensional cube. This can for instance be used to apply a digital elevation model to a spatio-temporal data cube. An overlap resolver is *required* to resolve the overlap for all pixels.

After the merge, the dimensions with a natural/inherent label order (with a reference system this is each spatial and temporal dimensions) still have all dimension labels sorted. For other dimensions where there is no inherent order, including bands, the

dimension labels keep the order in which they are present in the original data cubes and the dimension labels of `cube2` are appended to the dimension labels of `cube1`.

Parameters

cube1*

The first data cube.

> Data type: **raster-cube**

cube2*

The second data cube.

> Data type: **raster-cube**

overlap_resolver = null

A reduction operator that resolves the conflict if the data overlaps. The reducer must return a value of the same data type as the input values are. The reduction operator may be a single process such as `multiply` or consist of multiple sub-processes. `null` (the default) can be specified if no overlap resolver is required.

> Data type: **User-defined Process (process-graph:object)**
>
> **Parameters:**
>
> #### x*
>
> The overlapping value from the first data cube `cube1`.
>
> > Any data type.
> >
> > Data type: **any**
>
> #### y*
>
> The overlapping value from the second data cube `cube2`.
>
> > Any data type.
> >
> > Data type: **any**
>
> #### context = null
>
> Additional data passed by the user.
>
> > Any data type.
> >
> > Data type: **any**
>
> **Expected Return Value:**
>
> The value to be set in the merged data cube.

> Any data type.
>
> Data type: **any**

`context = null`

Additional data to be passed to the overlap resolver.

> Any data type.
>
> Data type: **any**

Return Value

The merged data cube. See the process description for details regarding the dimensions and dimension properties (name, type, labels, reference system and resolution).

> Data type: **raster-cube**

Errors/Exceptions

- `OverlapResolverMissing`
  Message: *Overlapping data cubes, but no overlap resolver has been specified.*

See Also

- Background information on reduction operators (binary reducers) by Wikipedia

min

Minimum value

MATH MATH > STATISTICS REDUCER Download JSON

Description

```
min(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

Computes the smallest value of an array of numbers, which is equal to the last element of a sorted (i.e., ordered) version of the array.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The minimum value.

> Data type: **number, null**

Examples

Example #1

```
min(data = [1,0,3,2]) => 0
```

Example #2

```
min(data = [5,2.5,null,-0.7]) => -0.7
```

Example #3

```
min(data = [1,0,3,null,2], ignore_nodata = false) => null
```

Example #4

```
min(data = []) => null
```

See Also

- Minimum explained by Wolfram MathWorld

mod

Modulo

MATH

Download JSON

Description

`mod(number|null x, number|null y) : number|null`

Remainder after a division of `x` by `y` for both integers and floating-point numbers.

The result of a modulo operation has the sign of the divisor. The handling regarding the sign of the result differs between programming languages and needs careful consideration to avoid unexpected results.

The no-data value `null` is passed through and therefore gets propagated if any of the arguments is `null`. A modulo by zero results in ±infinity if the processing environment supports it. Otherwise, a `DivisionByZero` exception must the thrown.

Parameters

x*

A number to be used as the dividend.

> Data type: **number, null**

y*

A number to be used as the divisor.

> Data type: **number, null**

Return Value

The remainder after division.

> Data type: **number, null**

Errors/Exceptions

- `DivisionByZero`
  Message: *Division by zero is not supported.*

Examples

Example #1

```
mod(x = 27, y = 5) => 2
```

Example #2

```
mod(x = -27, y = 5) => 3
```

Example #3

```
mod(x = 3.14, y = -2) => -0.86
```

Example #4

```
mod(x = -27, y = -5) => -2
```

Example #5

```
mod(x = 27, y = null) => null
```

Example #6

```
mod(x = null, y = 5) => null
```

See Also

- Modulo explained by Wikipedia

multiply

Multiplication of two numbers

MATH

Download JSON

Description

`multiply(number|null x, number|null y) : number|null`

Multiplies the two numbers `x` and `y` (*`x * y`*) and returns the computed product.

No-data values are taken into account so that `null` is returned if any element is such a value.

The computations follow IEEE Standard 754 whenever the processing environment supports it.

Parameters

x*

The multiplier.

Data type: **number, null**

y*

The multiplicand.

Data type: **number, null**

Return Value

The computed product of the two numbers.

> Data type: **number, null**

Errors/Exceptions

- `MultiplicandMissing`
  Message: *Multiplication requires at least two numbers.*

Examples

Example #1

```
multiply(x = 5, y = 2.5) => 12.5
```

Example #2

```
multiply(x = -2, y = -4) => 8
```

Example #3

```
multiply(x = 1, y = null) => null
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Product explained by Wolfram MathWorld

nan

Not a Number (NaN) — **experimental**

MATH > CONSTANTS

Download JSON

Description

```
nan() : any
```

`NaN` (not a number) is a symbolic floating-point representation which is neither a signed infinity nor a finite number.

Experimental
Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

This process has no parameters.

Return Value

Returns `NaN`.

> JSON Schema can't represent `NaN` and thus a schema can't be specified.
>
> Data type: **any**

See Also

- IEEE Standard 754-2008 for Floating-Point Arithmetic
- NaN explained by Wolfram MathWorld

ndvi

Normalized Difference Vegetation Index

MATH > INDICES VEGETATION INDICES Download JSON

Description

```
ndvi(raster-cube data, ?band-name:string nir = "nir", ?band-name:string red = "red", ?string|null target_band = null) : raster-cube
```

Computes the Normalized Difference Vegetation Index (NDVI). The NDVI is computed as *`(nir - red) / (nir + red)`*.

The `data` parameter expects a raster data cube with a dimension of type `bands` or a `DimensionAmbiguous` exception is thrown otherwise. By default, the dimension must have at least two bands with the common names `red` and `nir` assigned. Otherwise, the user has to specify the parameters `nir` and `red`. If neither is the case, either the exception `NirBandAmbiguous` or `RedBandAmbiguous` is thrown. The common names for each band are specified in the collection's band metadata and are *not* equal to the band names.

By default, the dimension of type `bands` is dropped by this process. To keep the dimension specify a new band name in the parameter `target_band`. This adds a new dimension label with the specified name to the dimension, which can be used to access the computed values. If a band with the specified name exists, a `BandExists` is thrown.

This process is very similar to the process `normalized_difference`, but determines the bands automatically based on the common names (`red`/`nir`) specified in the metadata.

Parameters

`data`*

A raster data cube with two bands that have the common names `red` and `nir` assigned.

> Data type: **raster-cube**

`nir` = `"nir"`

The name of the NIR band. Defaults to the band that has the common name `nir` assigned.

Either the unique band name (metadata field `name` in bands) or one of the common band names (metadata field `common_name` in bands) can be specified. If the unique band name and the common name conflict, the unique band name has a higher priority.

> Data type: **band-name:string**

`red` = `"red"`

The name of the red band. Defaults to the band that has the common name `red` assigned.

Either the unique band name (metadata field `name` in bands) or one of the common band names (metadata field `common_name` in bands) can be specified. If the unique band name and the common name conflict, the unique band name has a higher priority.

> Data type: **band-name:string**

`target_band` = `null`

By default, the dimension of type `bands` is dropped. To keep the dimension specify a new band name in this parameter so that a new dimension label with the specified name will be added for the computed values.

Data Types:

> Data type: **string**
>
> Pattern: `^\w+$`

> Data type: **null**

Return Value

A raster data cube containing the computed NDVI values. The structure of the data cube differs depending on the value passed to `target_band`:

- `target_band` is `null`: The data cube does not contain the dimension of type `bands`, the number of dimensions decreases by one. The dimension properties (name, type, labels, reference system and resolution) for all other dimensions remain unchanged.
- `target_band` is a string: The data cube keeps the same dimensions. The dimension properties remain unchanged, but the number of dimension labels for the dimension of type `bands` increases by one. The additional label is named as specified in `target_band`.

> Data type: **raster-cube**

Errors/Exceptions

- `NirBandAmbiguous`
  Message: *The NIR band can't be resolved, please specify the specific NIR band name.*
- `RedBandAmbiguous`

Message: *The red band can't be resolved, please specify the specific red band name.*

- `DimensionAmbiguous`

  Message: *dimension of type `bands` is not available or is ambiguous..*

- `BandExists`

  Message: *A band with the specified target name exists.*

See Also

- List of common band names as specified by the STAC specification
- NDVI explained by NASA
- NDVI explained by Wikipedia

neq

Not equal to comparison

TEXTS COMPARISON

Download JSON

Description

```
neq(any x, any y, ?number|null delta = null, ?boolean case_sensitive = true) : boolean|null
```

Compares whether `x` is *not* strictly equal to `y`.

Remarks:

- Data types MUST be checked strictly. For example, a string with the content *1* is not equal to the number *1*. Nevertheless, an integer *1* is equal to a floating-point number *1.0* as `integer` is a sub-type of `number`.
- If any operand is `null`, the return value is `null`. Therefore, `neq(null, null)` returns `null` instead of `false`.
- If any operand is an array or object, the return value is `false`.
- Strings are expected to be encoded in UTF-8 by default.
- Temporal strings MUST be compared differently than other strings and MUST NOT be compared based on their string representation due to different possible representations. For example, the time zone representation `Z` (for UTC) has the same meaning as `+00:00`.

Parameters

x*

First operand.

> Any data type is allowed.
>
> Data type: **any**

y*

Second operand.

> Any data type is allowed.
>
> Data type: **any**

`delta = null`

Only applicable for comparing two numbers. If this optional parameter is set to a positive non-zero number the non-equality of two numbers is checked against a delta value. This is especially useful to circumvent problems with floating-point inaccuracy in machine-based computation.

This option is basically an alias for the following computation: `gt(abs(minus([x, y]), delta)`

> Data type: **number, null**

`case_sensitive = true`

Only applicable for comparing two strings. Case sensitive comparison can be disabled by setting this parameter to `false`.

> Data type: **boolean**

Return Value

`true` if `x` is *not* equal to `y`, `null` if any operand is `null`, otherwise `false`.

> Data type: **boolean, null**

Examples

Example #1

```
neq(x = 1, y = null) => null
```

Example #2

```
neq(x = 1, y = 1) => false
```

Example #3

```
neq(x = 1, y = "1") => true
```

Example #4

```
neq(x = 0, y = false) => true
```

Example #5

```
neq(x = 1.02, y = 1, delta = 0.01) => true
```

Example #6

```
neq(x = -1, y = -1.001, delta = 0.01) => false
```

Example #7

```
neq(x = 115, y = 110, delta = 10) => false
```

Example #8

```
neq(x = "Test", y = "test") => true
```

Example #9

```
neq(x = "Test", y = "test", case_sensitive = false) => false
```

Example #10

```
neq(x = "Ä", y = "ä", case_sensitive = false) => false
```

Example #11

```
neq(x = "00:00:00+00:00", y = "00:00:00Z") => false
```

Example #12

y is not a valid date-time representation and therefore will be treated as a string so that the provided values are not equal.

```
neq(x = "2018-01-01T12:00:00Z", y = "2018-01-01T12:00:00") => true
```

Example #13

01:00 in the time zone +1 is equal to 00:00 in UTC.

```
neq(x = "2018-01-01T00:00:00Z", y = "2018-01-01T01:00:00+01:00") => false
```

Example #14

```
neq(x = [1,2,3], y = [1,2,3]) => false
```

normalized_difference

Normalized difference

MATH > INDICES VEGETATION INDICES

Download JSON

Description

```
normalized_difference(number x, number y) : number
```

Computes the normalized difference for two bands. The normalized difference is computed as `(x - y) / (x + y)`.

This process could be used for a number of remote sensing indices such as:

- NDVI: `x` = NIR band, `y` = red band
- NDWI: `x` = NIR band, `y` = SWIR band
- NDSI: `x` = green band, `y` = SWIR band

Some back-ends may have native processes such as `ndvi` available for convenience.

Parameters

x*

The value for the first band.

> Data type: **number**

y*

The value for the second band.

> Data type: **number**

Return Value

The computed normalized difference.

> Data type: **number**
>
> Minimum value (inclusive): -1
>
> Maximum value (inclusive): 1

See Also

- NDSI explained by EOS
- NDVI explained by EOS
- NDWI explained by EOS

not

Inverting a boolean

LOGIC

Download JSON

Description

```
not(boolean|null x) : boolean|null
```

Inverts a single boolean so that `true` gets `false` and `false` gets `true`.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

Boolean value to invert.

> Data type: **boolean, null**

Return Value

Inverted boolean value.

> Data type: **boolean, null**

Examples

Example #1

```
not(x = null) => null
```

Example #2

```
not(x = false) => true
```

Example #3

```
not(x = true) => false
```

or

Logical OR

LOGIC

Download JSON

Description

```
or(boolean|null x, boolean|null y) : boolean|null
```

Checks if **at least one** of the values is true. Evaluates parameter `x` before `y` and stops once the outcome is unambiguous. If a component is `null`, the result will be `null` if the outcome is ambiguous.

Truth table:

```
a \ b || null | false | true
----- || ---- | ----- | ----
null  || null | null  | true
false || null | false | true
true  || true | true  | true
```

Parameters

x*

A boolean value.

> Data type: **boolean, null**

y*

A boolean value.

> Data type: **boolean, null**

Return Value

Boolean result of the logical OR.

> Data type: **boolean, null**

Examples

Example #1

```
or(x = true, y = true) => true
```

Example #2

```
or(x = false, y = false) => false
```

Example #3

```
or(x = true, y = null) => true
```

Example #4

```
or(x = null, y = true) => true
```

Example #5

```
or(x = false, y = null) => null
```

order

Create a permutation

ARRAYS SORTING

Download JSON

Description

```
order(array<number|null|string> data, ?boolean asc = true, ?boolean|null nodata = null) : array<integer>
```

Computes a permutation which allows rearranging the data into ascending or descending order. In other words, this process computes the ranked (sorted) element positions in the original list.

Remarks:

- The positions in the result are zero-based.
- Ties will be left in their original ordering.
- Temporal strings can *not* be compared based on their string representation due to the time zone/time-offset representations.

Parameters

`data`*

An array to compute the order for.

> Data type: **array<number|null|date-time:string|date:string|time:string>**
>
> Array items: Data type: **any**

`asc = true`

The default sort order is ascending, with smallest values first. To sort in reverse (descending) order, set this parameter to `false`.

> Data type: **boolean**

`nodata = null`

Controls the handling of no-data values (`null`). By default, they are removed. If set to `true`, missing values in the data are put last; if set to `false`, they are put first.

> Data type: **boolean, null**

Return Value

The computed permutation.

> Data type: **array<integer>**

Array items:	Data type:	**integer**
	Minimum value (inclusive):	0

Examples

Example #1

```
order(data = [6,-1,2,null,7,4,null,8,3,9,9]) => [1,2,8,5,0,4,7,9,10]
```

Example #2

```
order(data = [6,-1,2,null,7,4,null,8,3,9,9], nodata = true) =>
[1,2,8,5,0,4,7,9,10,3,6]
```

Example #3

```
order(data = [6,-1,2,null,7,4,null,8,3,9,9], asc = false, nodata = true) =>
[9,10,7,4,0,5,8,2,1,3,6]
```

Example #4

```
order(data = [6,-1,2,null,7,4,null,8,3,9,9], asc = false, nodata = false) =>
[3,6,9,10,7,4,0,5,8,2,1]
```

See Also

- Permutation explained by Wolfram MathWorld

pi

Pi (π)

MATH > CONSTANTS MATH > TRIGONOMETRIC Download JSON

Description

```
pi() : number
```

The real number Pi (π) is a mathematical constant that is the ratio of the circumference of a circle to its diameter. The numerical value is approximately *3.14159*.

Parameters

This process has no parameters.

Return Value

The numerical value of Pi.

> Data type: **number**

See Also

- Mathematical constant Pi explained by Wolfram MathWorld

power

Exponentiation

MATH MATH > EXPONENTIAL & LOGARITHMIC Download JSON

Description

`power(number|null base, number|null p) : number|null`

Computes the exponentiation for the base `base` raised to the power of `p`.

The no-data value `null` is passed through and therefore gets propagated if any of the arguments is `null`.

Parameters

base*

The numerical base.

> Data type: **number, null**

p*

The numerical exponent.

> Data type: **number, null**

Return Value

The computed value for `base` raised to the power of `p`.

> Data type: **number, null**

Examples

Example #1

```
power(base = 0, p = 2) => 0
```

Example #2

```
power(base = 2.5, p = 0) => 1
```

Example #3

```
power(base = 3, p = 3) => 27
```

Example #4

```
power(base = 5, p = -1) => 0.2
```

Example #5

```
power(base = 1, p = 0.5) => 1
```

Example #6

```
power(base = 1, p = null) => null
```

Example #7

```
power(base = null, p = 2) => null
```

See Also

- Power explained by Wolfram MathWorld

predict_curve

Predict values — **experimental**

CUBES MATH

Download JSON

Description

```
predict_curve(raster-cube data, raster-cube parameters, process-graph:object function, string dimension, ?null|array<number|string> labels = null) : raster-cube
```

Predict values using a model function and pre-computed parameters. The process is primarily intended to compute values for new labels, but it can also fill gaps where existing labels contain no-data (`null`) values.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

data*

A data cube to predict values for.

> Data type: **raster-cube**

parameters*

A data cube with optimal values from a result of e.g. `fit_curve`.

> Data type: **raster-cube**

function*

The model function. It must take the parameters to fit as array through the first argument and the independent variable `x` as the second argument.

It is recommended to store the model function as a user-defined process on the back-end.

> Data type: **User-defined Process (process-graph:object)**
>
> **Parameters:**
>
> #### x*
>
> The value for the independent variable `x`.
>
> > Data type: **number**
>
> #### parameters*
>
> The parameters for the model function, contains at least one parameter.
>
> > Data type: **array<number>**
> >
> > Min. number of items: 1
> >
> > Array items:
> >
> > > Data type: **number**
>
> **Expected Return Value:**
>
> The computed value `y` value for the given independent variable `x` and the parameters.
>
> > Data type: **number**

dimension*

The name of the dimension for predictions. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

> Data type: **string**

`labels` = `null`

The labels to predict values for. If no labels are given, predicts values only for no-data (`null`) values in the data cube.

Data Types:

> Data type: **null**

> Data type: **array<number|date:string|date-time:string>**
>
> Array items: Data type: **any**

Return Value

A data cube with the predicted values.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

product

Compute the product by multiplying numbers

MATH REDUCER

Download JSON

Description

```
product(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

Multiplies all elements in a sequential array of numbers and returns the computed product.

By default no-data values are ignored. Setting `ignore_nodata` to `false` considers no-data values so that `null` is returned if any element is such a value.

The computations follow IEEE Standard 754 whenever the processing environment supports it.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The computed product of the sequence of numbers.

> Data type: **number, null**

Examples

Example #1

```
product(data = [5,0]) => 0
```

Example #2

```
product(data = [-2,4,2.5]) => -20
```

Example #3

```
product(data = [1,null], ignore_nodata = false) => null
```

Example #4

```
product(data = [-1]) => -1
```

Example #5

```
product(data = [null], ignore_nodata = false) => null
```

Example #6

```
product(data = []) => null
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Product explained by Wolfram MathWorld

quantiles

Quantiles

MATH > STATISTICS

Download JSON

Description

```
quantiles(array<number|null> data, ?array<number> probabilities, ?integer q, ?boolean ignore_nodata = true) : array<number|null>
```

Calculates quantiles, which are cut points dividing the range of a sample distribution into either

- intervals corresponding to the given `probabilities` or
- equal-sized intervals (q-quantiles based on the parameter `q`).

Either the parameter `probabilities` or `q` must be specified, otherwise the `QuantilesParameterMissing` exception is thrown. If both parameters are set the `QuantilesParameterConflict` exception is thrown.

Sample quantiles can be computed with several different algorithms. Hyndman and Fan (1996) have concluded on nine different types, which are commonly implemented in statistical software packages. This process is implementing type 7, which is implemented widely and often also the default type (e.g. in Excel, Julia, Python, R and S).

Parameters

data*

An array of numbers.

Data type: **array<number|null>**

probabilities

A list of probabilities to calculate quantiles for. The probabilities must be between 0 and 1 (inclusive).

Data type: **array<number>**

Array items:

- Data type: **number**
- Minimum value (inclusive): 0
- Maximum value (inclusive): 1

q

Number of intervals to calculate quantiles for. Calculates q-quantiles with equal-sized intervals.

Data type: **integer**

Minimum value (inclusive): 2

ignore_nodata = true

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that an array with `null` values is returned if any element is such a value.

> Data type: **boolean**

Return Value

An array with the computed quantiles. The list has either

- as many elements as the given list of `probabilities` had or
- *`q`-1* elements.

If the input array is empty the resulting array is filled with as many `null` values as required according to the list above. See the 'Empty array' example for an example.

> Data type: **array<number|null>**

Errors/Exceptions

- `QuantilesParameterMissing`
  Message: *The process `quantiles` requires either the `probabilities` or `q` parameter to be set.*
- `QuantilesParameterConflict`
  Message: *The process `quantiles` only allows that either the `probabilities` or the `q` parameter is set.*

Examples

Example #1

```
quantiles(data = [2,4,4,4,5,5,7,9], probabilities = [0.005,0.01,0.02,0.05,0.1,0.5]) => [2.07,2.14,2.28,2.7,3.4,4.5]
```

Example #2

```
quantiles(data = [2,4,4,4,5,5,7,9], q = 4) => [4,4.5,5.5]
```

Example #3

```
quantiles(data = [-1,-0.5,null,1], q = 2) => [-0.5]
```

Example #4

```
quantiles(data = [-1,-0.5,null,1], q = 4, ignore_nodata = false) => [null,null,null]
```

Empty array (#5)

```
quantiles(data = [], probabilities = [0.1,0.5]) => [null,null]
```

See Also

- Hyndman and Fan (1996): Sample Quantiles in Statistical Packages

- Quantiles explained by Wikipedia

rearrange

Rearrange an array based on a permutation

ARRAYS SORTING

Download JSON

Description

```
rearrange(array data, array<integer> order) : array
```

Rearranges an array based on a permutation, i.e. a ranked list of element positions in the original list. The positions must be zero-based.

Parameters

data*

The array to rearrange.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

order*

The permutation used for rearranging.

Data type: **array<integer>**

Array items:

Data type: **integer**

Minimum value (inclusive): 0

Return Value

The rearranged array.

Data type: **array**

Array items:

Any data type is allowed.

Data type: **any**

Examples

Reverse a list (#1)

```
rearrange(data = [5,4,3], order = [2,1,0]) => [3,4,5]
```

Remove two elements (#2)

```
rearrange(data = [5,4,3,2], order = [1,3]) => [4,2]
```

Swap two elements (#3)

```
rearrange(data = [5,4,3,2], order = [0,2,1,3]) => [5,3,4,2]
```

See Also

- Permutation explained by Wolfram MathWorld

reduce_dimension

Reduce dimensions

CUBES REDUCER

Download JSON

Description

```
reduce_dimension(raster-cube data, process-graph:object reducer, string dimension, ?any context = null) : raster-cube
```

Applies a reducer to a data cube dimension by collapsing all the pixel values along the specified dimension into an output value computed by the reducer.

The dimension is dropped. To avoid this, use `apply_dimension` instead.

Parameters

`data`*

A data cube.

Data type: **raster-cube**

`reducer`*

A reducer to apply on the specified dimension. A reducer is a single process such as `mean` or a set of processes, which computes a single value for a list of values, see the category 'reducer' for such processes.

Data type: **User-defined Process (process-graph:object)**

Parameters:

`data`*

A labeled array with elements of any type.

Data type: **labeled-array**

Array items:

Any data type.

Data type: **any**

`context` = `null`

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new data cube.

Any data type.

Data type: **any**

`dimension`*

The name of the dimension over which to reduce. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

Data type: **string**

`context` = `null`

Additional data to be passed to the reducer.

Any data type.

Data type: **any**

Return Value

A data cube with the newly computed values. It is missing the given dimension, the number of dimensions decreases by one. The dimension properties (name, type, labels, reference system and resolution) for all other dimensions remain unchanged.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

See Also

- Reducers explained in the openEO documentation

reduce_spatial

Reduce spatial dimensions 'x' and 'y' — **experimental**

AGGREGATE & RESAMPLE CUBES REDUCER Download JSON

Description

```
reduce_spatial(raster-cube data, process-graph:object reducer, ?any context = null) : raster-cube
```

Applies a reducer to a data cube by collapsing all the pixel values along the horizontal spatial dimensions (i.e. axes `x` and `y`) into an output value computed by the reducer. The horizontal spatial dimensions are dropped.

An aggregation over certain spatial areas can be computed with the process `aggregate_spatial`.

This process passes a list of values to the reducer. The list of values has an undefined order, therefore processes such as `last` and `first` that depend on the order of the values will lead to unpredictable results.

Experimental
Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`reducer`*

A reducer to apply on the horizontal spatial dimensions. A reducer is a single process such as `mean` or a set of processes, which computes a single value for a list of values, see the category 'reducer' for such processes.

> Data type: **User-defined Process (process-graph:object)**

Parameters:

`data`*

An array with elements of any type.

Data type: **array**

Array items:

Any data type.

Data type: **any**

`context` = `null`

Additional data passed by the user.

Any data type.

Data type: **any**

Expected Return Value:

The value to be set in the new data cube.

Any data type.

Data type: **any**

`context` = `null`

Additional data to be passed to the reducer.

Any data type.

Data type: **any**

Return Value

A data cube with the newly computed values. It is missing the horizontal spatial dimensions, the number of dimensions decreases by two. The dimension properties (name, type, labels, reference system and resolution) for all other dimensions remain unchanged.

Data type: **raster-cube**

See Also

- Reducers explained in the openEO documentation

rename_dimension

Rename a dimension

CUBES

Download JSON

Description

```
rename_dimension(raster-cube data, string source, string target) : raster-cube
```

Renames a dimension in the data cube while preserving all other properties.

Parameters

`data`*

The data cube.

> Data type: **raster-cube**

`source`*

The current name of the dimension. Fails with a `DimensionNotAvailable` exception if the specified dimension does not exist.

> Data type: **string**

`target`*

A new Name for the dimension. Fails with a `DimensionExists` exception if a dimension with the specified name exists.

> Data type: **string**

Return Value

A data cube with the same dimensions, but the name of one of the dimensions changes. The old name can not be referred to any longer. The dimension properties (name, type, labels, reference system and resolution) remain unchanged.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*
- `DimensionExists`
  Message: *A dimension with the specified name already exists.*

rename_labels

Rename dimension labels

CUBES

Download JSON

Description

```
rename_labels(raster-cube data, string dimension, array<number|string> target, ?
array<number|string> source = []) : raster-cube
```

Renames the labels of the specified dimension in the data cube from `source` to `target`.

If the array for the source labels is empty (the default), the dimension labels are expected to be enumerated with zero-based numbering (0,1,2,3,...) so that the dimension labels directly map to the indices of the array specified for the parameter `target`. If the dimension labels are not enumerated and the `target` parameter is not specified, the `LabelsNotEnumerated` exception is thrown. The number of the source and target labels must be equal. Otherwise, the exception `LabelMismatch` is thrown.

This process doesn't change the order of the labels and their corresponding data.

Parameters

data*

The data cube.

> Data type: **raster-cube**

dimension*

The name of the dimension to rename the labels for.

> Data type: **string**

target*

The new names for the labels. The dimension labels in the data cube are expected to be enumerated if the parameter `target` is not specified. If a target dimension label already exists in the data cube, a `LabelExists` exception is thrown.

> Data type: **array<number|string>**

source = []

The names of the labels as they are currently in the data cube. The array defines an unsorted and potentially incomplete list of labels that should be renamed to the names available in the corresponding array elements in the parameter `target`. If one of the source dimension labels doesn't exist, the `LabelNotAvailable` exception is thrown. By default, the array is empty so that the dimension labels in the data cube are expected to be enumerated.

> Data type: **array<number|string>**

Return Value

The data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged, except that for the given dimension the labels change. The old labels can not be referred to any longer. The number of labels remains the same.

> Data type: **raster-cube**

Errors/Exceptions

- `LabelsNotEnumerated`
  Message: *The dimension labels are not enumerated.*
- `LabelMismatch`
  Message: *The number of labels in the parameters `source` and `target` don't match.*
- `LabelNotAvailable`
  Message: *A label with the specified name does not exist.*
- `LabelExists`
  Message: *A label with the specified name exists.*

Examples

Rename named labels (#1)

> Renaming the bands from `B1` to `red`, from `B2` to `green` and from `B3` to `blue`.
>
> ```
> rename_labels(data = $data, dimension = "bands", target = ["red","green","blue"],
> source = ["B1","B2","B3"])
> ```

Processes

- Rename enumerated labels

resample_cube_spatial

Resample the spatial dimensions to match a target data cube

CUBES AGGREGATE & RESAMPLE

Download JSON

Description

```
resample_cube_spatial(raster-cube data, raster-cube target, ?string method = "near") :
raster-cube
```

Resamples the spatial dimensions (x,y) from a source data cube to align with the corresponding dimensions of the given target data cube. Returns a new data cube with the resampled dimensions.

To resample a data cube to a specific resolution or projection regardless of an existing target data cube, refer to `resample_spatial`.

Parameters

`data`*

A data cube.

> Data type: **raster-cube**

`target`*

A data cube that describes the spatial target resolution.

> Data type: **raster-cube**

`method = "near"`

Resampling method to use. The following options are available and are meant to align with `gdalwarp`:

- `average`: average (mean) resampling, computes the weighted average of all valid pixels
- `bilinear`: bilinear resampling
- `cubic`: cubic resampling
- `cubicspline`: cubic spline resampling
- `lanczos`: Lanczos windowed sinc resampling
- `max`: maximum resampling, selects the maximum value from all valid pixels
- `med`: median resampling, selects the median value of all valid pixels
- `min`: minimum resampling, selects the minimum value from all valid pixels
- `mode`: mode resampling, selects the value which appears most often of all the sampled points
- `near`: nearest neighbour resampling (default)
- `q1`: first quartile resampling, selects the first quartile value of all valid pixels
- `q3`: third quartile resampling, selects the third quartile value of all valid pixels
- `rms` root mean square (quadratic mean) of all valid pixels
- `sum`: compute the weighted sum of all valid pixels

Valid pixels are determined based on the function `is_valid`.

> Data type: **string**
>
> Allowed values: average, bilinear, cubic, cubicspline, lanczos, max, med, min, mode, near, q1, q3, rms, sum

Return Value

A data cube with the same dimensions. The dimension properties (name, type, labels, reference system and resolution) remain unchanged, except for the resolution and dimension labels of the spatial dimensions.

> Data type: **raster-cube**

See Also

- Resampling explained in the openEO documentation

resample_cube_temporal

Resample temporal dimensions to match a target data cube — **experimental**

CUBES AGGREGATE & RESAMPLE

Download JSON

Description

```
resample_cube_temporal(raster-cube data, raster-cube target, ?string|null dimension = null, ?number|null valid_within = null) : raster-cube
```

Resamples one or more given temporal dimensions from a source data cube to align with the corresponding dimensions of the given target data cube using the nearest neighbor method. Returns a new data cube with the resampled dimensions.

By default, this process simply takes the nearest neighbor independent of the value (including values such as no-data / `null`). Depending on the data cubes this may lead to values being assigned to two target timestamps. To only consider valid values in a specific range around the target timestamps, use the parameter `valid_within`.

The rare case of ties is resolved by choosing the earlier timestamps.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

data*

A data cube with one or more temporal dimensions.

> Data type: **raster-cube**

target*

A data cube that describes the temporal target resolution.

> Data type: **raster-cube**

dimension = null

The name of the temporal dimension to resample, which must exist with this name in both data cubes. If the dimension is not set or is set to `null`, the process resamples all temporal dimensions that exist with the same names in both data cubes.

The following exceptions may occur:

- A dimension is given, but it does not exist in any of the data cubes: `DimensionNotAvailable`
- A dimension is given, but one of them is not temporal: `DimensionMismatch`
- No specific dimension name is given and there are no temporal dimensions with the same name in the data: `DimensionMismatch`

> Data type: **string, null**

`valid_within` = `null`

Setting this parameter to a numerical value enables that the process searches for valid values within the given period of days before and after the target timestamps. Valid values are determined based on the function `is_valid`. For example, the limit of `7` for the target timestamps `2020-01-15 12:00:00` looks for a nearest neighbor after `2020-01-08 12:00:00` and before `2020-01-22 12:00:00`. If no valid value is found within the given period, the value will be set to no-data (`null`).

> Data type: **number, null**

Return Value

A raster data cube with the same dimensions and the same dimension properties (name, type, labels, reference system and resolution) for all non-temporal dimensions. For the temporal dimension, the name and type remain unchanged, but the dimension labels, resolution and reference system may change.

> Data type: **raster-cube**

Errors/Exceptions

- `DimensionMismatch`
  Message: *The temporal dimensions for resampling don't match.*
- `DimensionNotAvailable`
  Message: *A dimension with the specified name does not exist.*

See Also

- Resampling explained in the openEO documentation

resample_spatial

Resample and warp the spatial dimensions

CUBES AGGREGATE & RESAMPLE

Download JSON

Description

```
resample_spatial(raster-cube data, ?number|array<number> resolution = 0, ?epsg-
code:integer|string|null projection = null, ?string method = "near", ?string align = "upper-
left") : raster-cube
```

Resamples the spatial dimensions (x,y) of the data cube to a specified resolution and/or warps the data cube to the target projection. At least `resolution` or `projection` must be specified.

Related processes:

- Use `filter_bbox` to set the target spatial extent.
- To spatially align two data cubes with each other (e.g. for merging), better use the process `resample_cube_spatial`.

Parameters

`data`*

A raster data cube.

> Data type: **raster-cube**

`resolution = 0`

Resamples the data cube to the target resolution, which can be specified either as separate values for x and y or as a single value for both axes. Specified in the units of the target projection. Doesn't change the resolution by default (`0`).

Data Types:

> A single number used as the resolution for both x and y.
>
> Data type: **number**
>
> Minimum value (inclusive): 0

> A two-element array to specify separate resolutions for x (first element) and y (second element).
>
> Data type: **array<number>**
>
> Min. number of items: 2
>
> Max. number of items: 2
>
> Array items:
>
> > Data type: **number**
> >
> > Minimum value (inclusive): 0

`projection = null`

Warps the data cube to the target projection, specified as as EPSG code, WKT2 (ISO 19162) string, PROJ definition (deprecated). By default (`null`), the projection is not changed.

Data Types:

> **EPSG Code**
>
> Data type: **epsg-code:integer**
>
> Minimum value (inclusive): 1000
>
> Examples: 3857

> **WKT2**

Data type: **wkt2-definition:string**

PROJ definition

Data type: **proj-definition:string**

Deprecated: ✓ Yes

Don't change projection

Data type: **null**

`method = "near"`

Resampling method to use. The following options are available and are meant to align with `gdalwarp`:

- `average`: average (mean) resampling, computes the weighted average of all valid pixels
- `bilinear`: bilinear resampling
- `cubic`: cubic resampling
- `cubicspline`: cubic spline resampling
- `lanczos`: Lanczos windowed sinc resampling
- `max`: maximum resampling, selects the maximum value from all valid pixels
- `med`: median resampling, selects the median value of all valid pixels
- `min`: minimum resampling, selects the minimum value from all valid pixels
- `mode`: mode resampling, selects the value which appears most often of all the sampled points
- `near`: nearest neighbour resampling (default)
- `q1`: first quartile resampling, selects the first quartile value of all valid pixels
- `q3`: third quartile resampling, selects the third quartile value of all valid pixels
- `rms` root mean square (quadratic mean) of all valid pixels
- `sum`: compute the weighted sum of all valid pixels

Valid pixels are determined based on the function `is_valid`.

Data type: **string**

Allowed values: average, bilinear, cubic, cubicspline, lanczos, max, med, min, mode, near, q1, q3, rms, sum

`align = "upper-left"`

Specifies to which corner of the spatial extent the new resampled data is aligned to.

Data type: **string**

Allowed values: lower-left, upper-left, lower-right, upper-right

Return Value

A raster data cube with values warped onto the new projection. It has the same dimensions and the same dimension properties (name, type, labels, reference system and resolution) for all non-spatial or vertical spatial dimensions. For the horizontal spatial dimensions the name and type remain unchanged, but reference system, labels and resolution may change depending on the given parameters.

> Data type: **raster-cube**

See Also

- gdalwarp resampling methods
- Official EPSG code registry
- PROJ parameters for cartographic projections
- Resampling explained in the openEO documentation
- Unofficial EPSG code database

round

Round to a specified precision

MATH > ROUNDING

Download JSON

Description

```
round(number|null x, ?integer p = 0) : number|null
```

Rounds a real number `x` to specified precision `p`.

If the fractional part of `x` is halfway between two integers, one of which is even and the other odd, then the even number is returned. This behavior follows IEEE Standard 754. This kind of rounding is also called "round to nearest (even)" or "banker's rounding". It minimizes rounding errors that result from consistently rounding a midpoint value in a single direction.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number to round.

> Data type: **number, null**

p = 0

A positive number specifies the number of digits after the decimal point to round to. A negative number means rounding to a power of ten, so for example *-2* rounds to the nearest hundred. Defaults to *0*.

> Data type: **integer**

Return Value

The rounded number.

> Data type: **number, null**

Examples

Example #1

```
round(x = 0) => 0
```

Example #2

```
round(x = 3.56, p = 1) => 3.6
```

Example #3

```
round(x = -0.4444444, p = 2) => -0.44
```

Example #4

```
round(x = -2.5) => -2
```

Example #5

```
round(x = -3.5) => -4
```

Example #6

```
round(x = 1234.5, p = -2) => 1200
```

See Also

- Absolute value explained by Wolfram MathWorld
- IEEE Standard 754-2019 for Floating-Point Arithmetic

run_udf

Run a UDF

CUBES IMPORT UDF

Download JSON

Description

```
run_udf(any data, string udf, udf-runtime:string runtime, ?udf-runtime-version:string|null
version = null, ?object context = {}) : any
```

Runs a UDF in one of the supported runtime environments.

The process can either:

1. load and run a UDF stored in a file on the server-side workspace of the authenticated user. The path to the UDF file must be relative to the root directory of the user's workspace.
2. fetch and run a remotely stored and published UDF by absolute URI.

3. run the source code specified inline as string.

The loaded UDF can be executed in several processes such as `aggregate_spatial`, `apply`, `apply_dimension` and `reduce_dimension`. The user must ensure that the data is given in a way that the UDF code can make sense of it.

Parameters

`data`*

The data to be passed to the UDF.

Data Types:

> **Array**
>
> Data type: **array**
>
> Min. number of items: 1
>
> Array items:
>
> > Any data type.
> >
> > Data type: **any**

> **Single Value**
>
> A single value of any data type.
>
> Data type: **any**

`udf`*

Either source code, an absolute URL or a path to a UDF script.

Data Types:

> Absolute URL to a UDF
>
> Data type: **uri:string**
>
> Pattern: `^https?://`

> Path to a UDF uploaded to the server.
>
> Data type: **file-path:string**
>
> Pattern: `^[^\r\n\:'"]+$`

> The multi-line source code of a UDF, must contain a newline/line-break.
>
> Data type: **udf-code:string**
>
> Pattern: `(\r\n|\r|\n)`

`runtime`*

A UDF runtime identifier available at the back-end.

> Data type: **udf-runtime:string**

`version = null`

An UDF runtime version. If set to `null`, the default runtime version specified for each runtime is used.

Data Types:

> Data type: **udf-runtime-version:string**

> **Default runtime version**
>
> Data type: **null**

`context = {}`

Additional data such as configuration options to be passed to the UDF.

> Data type: **object**

Return Value

The data processed by the UDF. The returned value can be of any data type and is exactly what the UDF code returns.

> **Any**
>
> Any data type.
>
> Data type: **any**

Errors/Exceptions

- `InvalidRuntime`
  Message: *The specified UDF runtime is not supported.*
- `InvalidVersion`
  Message: *The specified UDF runtime version is not supported.*

run_udf_externally

Run an externally hosted UDF container — **experimental**

CUBES IMPORT UDF

Download JSON

Description

```
run_udf_externally(any data, uri:string url, ?object context = {}) : any
```

Runs a compatible UDF container that is either externally hosted by a service provider or running on a local machine of the user. The UDF container must follow the openEO UDF specification.

The referenced UDF service can be executed in several processes such as `aggregate_spatial`, `apply`, `apply_dimension` and `reduce_dimension`. In this case, an array is passed instead of a raster data cube. The user must ensure that the data is given in a way that the UDF code can make sense of it.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

The data to be passed to the UDF.

Data Types:

Array

Data type: **array**

Min. number of items: 1

Array items:

Any data type.

Data type: **any**

Single Value

A single value of any data type.

Data type: **any**

`url`*

Absolute URL to a remote UDF service.

> Data type: **uri:string**
>
> Pattern: `^https?://`

`context = {}`

Additional data such as configuration options to be passed to the UDF.

> Data type: **object**

Return Value

The data processed by the UDF. The returned value can in principle be of any data type, but it depends on what is returned by the UDF code. Please see the implemented UDF interface for details.

> **Any**
>
> Any data type.
>
> Data type: **any**

See Also

- openEO UDF repository
- openEO UDF specification

sar_backscatter

Computes backscatter from SAR input — **experimental**

CUBES SAR

Download JSON

Description

```
sar_backscatter(raster-cube data, ?string|null coefficient = "gamma0-terrain", ?collection-id:string|null elevation_model = null, ?boolean mask = false, ?boolean contributing_area = false, ?boolean local_incidence_angle = false, ?boolean ellipsoid_incidence_angle = false, ?boolean noise_removal = true, ?object options = {}) : raster-cube
```

Computes backscatter from SAR input.

Note that backscatter computation may require instrument specific metadata that is tightly coupled to the original SAR products. As a result, this process may only work in combination with loading data from specific collections, not with general data cubes.

This process uses bilinear interpolation, both for resampling the DEM and the backscatter.

Experimental

Please note that this process is experimental with the potential for major things to change. Feel encouraged to try it out and give feedback, but refrain from using it in production.

Parameters

`data`*

The source data cube containing SAR input.

> Data type: **raster-cube**

`coefficient` = `"gamma0-terrain"`

Select the radiometric correction coefficient. The following options are available:

- `beta0`: radar brightness
- `sigma0-ellipsoid`: ground area computed with ellipsoid earth model
- `sigma0-terrain`: ground area computed with terrain earth model
- `gamma0-ellipsoid`: ground area computed with ellipsoid earth model in sensor line of sight
- `gamma0-terrain`: ground area computed with terrain earth model in sensor line of sight (default)
- `null`: non-normalized backscatter

Data Types:

> Data type: **string**
>
> Allowed values: beta0, sigma0-ellipsoid, sigma0-terrain, gamma0-ellipsoid, gamma0-terrain

> **Non-normalized backscatter**
>
> Data type: **null**

`elevation_model` = `null`

The digital elevation model to use. Set to `null` (the default) to allow the back-end to choose, which will improve portability, but reduce reproducibility.

Data Types:

> Data type: **collection-id:string**

> Data type: **null**

`mask` = `false`

If set to `true`, a data mask is added to the bands with the name `mask`. It indicates which values are valid (1), invalid (0) or contain no-data (null).

> Data type: **boolean**

`contributing_area` = `false`

If set to `true`, a DEM-based local contributing area band named `contributing_area` is added. The values are given in square meters.

> Data type: **boolean**

`local_incidence_angle` = `false`

If set to `true`, a DEM-based local incidence angle band named `local_incidence_angle` is added. The values are given in degrees.

> Data type: **boolean**

`ellipsoid_incidence_angle` = `false`

If set to `true`, an ellipsoidal incidence angle band named `ellipsoid_incidence_angle` is added. The values are given in degrees.

> Data type: **boolean**

`noise_removal` = `true`

If set to `false`, no noise removal is applied. Defaults to `true`, which removes noise.

> Data type: **boolean**

`options` = `{}`

Proprietary options for the backscatter computations. Specifying proprietary options will reduce portability.

> Data type: **object**
>
> Each property: ✕ No

Return Value

Backscatter values corresponding to the chosen parametrization. The values are given in linear scale.

> Data type: **raster-cube**

Errors/Exceptions

- `DigitalElevationModelInvalid`
  Message: *The digital elevation model specified is either not a DEM or can't be used with the data cube given.*

See Also

- Flattening Gamma: Radiometric Terrain Correction for SAR Imagery
- Gamma nought (0) explained by EO4GEO body of knowledge.
- Reasoning behind the choice of bilinear resampling
- Sigma nought (0) explained by EO4GEO body of knowledge.

save_result

Save processed data

CUBES EXPORT

Download JSON

Description

```
save_result(raster-cube|vector-cube data, output-format:string format, ?output-format-options:object options = {}) : boolean
```

Makes the processed data available in the given file format to the corresponding medium that is relevant for the context this processes is applied in:

- For **batch jobs** the data is stored on the back-end. STAC-compatible metadata is usually made available with the processed data.
- For **synchronous processing** the data is sent to the client as a direct response to the request.
- **Secondary web services** are provided with the processed data so that it can make use of it (e.g., visualize it). Web service may require the data in a certain format. Please refer to the documentation of the individual service types for details.

Parameters

`data`*

The data to deliver in the given file format.

Data Types:

> Data type: **raster-cube**

> Data type: **vector-cube**

`format`*

The file format to use. It must be one of the values that the server reports as supported output file formats, which usually correspond to the short GDAL/OGR codes. If the format is not suitable for storing the underlying data structure, a `FormatUnsuitable` exception will be thrown. This parameter is *case insensitive*.

> Data type: **output-format:string**

`options = {}`

The file format parameters to be used to create the file(s). Must correspond to the parameters that the server reports as supported parameters for the chosen `format`. The parameter names and valid values usually correspond to the GDAL/OGR format options.

> Data type: **output-format-options:object**

Return Value

Returns `false` if the process failed to make the data available, `true` otherwise.

> Data type: **boolean**

Errors/Exceptions

- `FormatUnsuitable`
  Message: *Data can't be transformed into the requested output format.*

See Also

- GDAL Raster Formats
- OGR Vector Formats

sd

Standard deviation

MATH > STATISTICS REDUCER Download JSON

Description

```
sd(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

Computes the sample standard deviation, which quantifies the amount of variation of an array of numbers. It is defined to be the square root of the corresponding variance (see `variance`).

A low standard deviation indicates that the values tend to be close to the expected value, while a high standard deviation indicates that the values are spread out over a wider range.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The computed sample standard deviation.

> Data type: **number, null**

Examples

Example #1

```
sd(data = [-1,1,3,null]) => 2
```

Example #2

```
sd(data = [-1,1,3,null], ignore_nodata = false) => null
```

Example #3

> The input array is empty: return `null`.

```
sd(data = []) => null
```

See Also

- Standard deviation explained by Wolfram MathWorld

sgn

Signum

MATH

Download JSON

Description

```
sgn(number|null x) : number|null
```

The signum (also known as *sign*) of `x` is defined as:

- *1* if *x > 0*
- *0* if *x = 0*
- *-1* if *x < 0*

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed signum value of `x`.

> Data type: **number, null**

Examples

Example #1

```
sgn(x = -2) => -1
```

Example #2

```
sgn(x = 3.5) => 1
```

Example #3

```
sgn(x = 0) => 0
```

Example #4

```
sgn(x = null) => null
```

See Also

- Sign explained by Wolfram MathWorld

sin

Sine

MATH > TRIGONOMETRIC

Download JSON

Description

```
sin(number|null x) : number|null
```

Computes the sine of `x`.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

An angle in radians.

> Data type: **number, null**

Return Value

The computed sine of `x`.

> Data type: **number, null**

Examples

Example #1

```
sin(x = 0) => 0
```

See Also

- Sine explained by Wolfram MathWorld

sinh

Hyperbolic sine

MATH > TRIGONOMETRIC

Download JSON

Description

```
sinh(number|null x) : number|null
```

Computes the hyperbolic sine of `x`.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

An angle in radians.

> Data type: **number, null**

Return Value

The computed hyperbolic sine of `x`.

> Data type: **number, null**

Examples

Example #1

```
sinh(x = 0) => 0
```

See Also

- Hyperbolic sine explained by Wolfram MathWorld

sort

Sort data

ARRAYS SORTING

Download JSON

Description

```
sort(array<number|null|string> data, ?boolean asc = true, ?boolean|null nodata = null) :
array<number|null|string>
```

Sorts an array into ascending (default) or descending order.

Remarks:

- Ties will be left in their original ordering.
- Temporal strings can *not* be compared based on their string representation due to the time zone/time-offset representations.

Parameters

`data`*

An array with data to sort.

> Data type: **array<number|null|date-time:string|date:string|time:string>**
>
> Array items: Data type: **any**

`asc = true`

The default sort order is ascending, with smallest values first. To sort in reverse (descending) order, set this parameter to `false`.

> Data type: **boolean**

`nodata = null`

Controls the handling of no-data values (`null`). By default, they are removed. If set to `true`, missing values in the data are put last; if set to `false`, they are put first.

> Data type: **boolean, null**

Return Value

The sorted array.

> Data type: **array<number|null|date-time:string|date:string|time:string>**
>
> Array items: Data type: **any**

Examples

Example #1

```
sort(data = [6,-1,2,null,7,4,null,8,3,9,9]) => [-1,2,3,4,6,7,8,9,9]
```

Example #2

```
sort(data = [6,-1,2,null,7,4,null,8,3,9,9], asc = false, nodata = true) =>
[9,9,8,7,6,4,3,2,-1,null,null]
```

sqrt

Square root

MATH MATH > EXPONENTIAL & LOGARITHMIC Download JSON

Description

```
sqrt(number|null x) : number|null
```

Computes the square root of a real number `x`, which is equal to calculating `x` to the power of *0.5*.

A square root of x is a number a such that $a^2 = x$. Therefore, the square root is the inverse function of a to the power of 2, but only for $a >= 0$.

The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

A number.

> Data type: **number, null**

Return Value

The computed square root.

> Data type: **number, null**

Examples

Example #1

```
sqrt(x = 0) => 0
```

Example #2

```
sqrt(x = 1) => 1
```

Example #3

```
sqrt(x = 9) => 3
```

Example #4

```
sqrt(x = null) => null
```

See Also

- Square root explained by Wolfram MathWorld

subtract

Subtraction of two numbers

MATH

Download JSON

Description

```
subtract(number|null x, number|null y) : number|null
```

Subtracts argument `y` from the argument `x` (*`x - y`*) and returns the computed result.

No-data values are taken into account so that `null` is returned if any element is such a value.

The computations follow IEEE Standard 754 whenever the processing environment supports it.

Parameters

x*

The minuend.

> Data type: **number, null**

y*

The subtrahend.

> Data type: **number, null**

Return Value

The computed result.

> Data type: **number, null**

Examples

Example #1

```
subtract(x = 5, y = 2.5) => 2.5
```

Example #2

```
subtract(x = -2, y = 4) => -6
```

Example #3

```
subtract(x = 1, y = null) => null
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Subtraction explained by Wolfram MathWorld

sum

Compute the sum by adding up numbers

MATH REDUCER

Download JSON

Description

```
sum(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

Sums up all elements in a sequential array of numbers and returns the computed sum.

By default no-data values are ignored. Setting `ignore_nodata` to `false` considers no-data values so that `null` is returned if any element is such a value.

The computations follow IEEE Standard 754 whenever the processing environment supports it.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The computed sum of the sequence of numbers.

> Data type: **number, null**

Examples

Example #1

```
sum(data = [5,1]) => 6
```

Example #2

```
sum(data = [-2,4,2.5]) => 4.5
```

Example #3

```
sum(data = [1,null], ignore_nodata = false) => null
```

Example #4

```
sum(data = [100]) => 100
```

Example #5

```
sum(data = [null], ignore_nodata = false) => null
```

Example #6

```
sum(data = []) => null
```

See Also

- IEEE Standard 754-2019 for Floating-Point Arithmetic
- Sum explained by Wolfram MathWorld

tan

Tangent

MATH > TRIGONOMETRIC

Download JSON

Description

`tan(number|null x) : number|null`

Computes the tangent of `x`. The tangent is defined to be the sine of x divided by the cosine of x.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

An angle in radians.

> Data type: **number, null**

Return Value

The computed tangent of `x`.

> Data type: **number, null**

Examples

Example #1

```
tan(x = 0) => 0
```

See Also

- Tangent explained by Wolfram MathWorld

tanh

Hyperbolic tangent

MATH > TRIGONOMETRIC

Download JSON

Description

`tanh(number|null x) : number|null`

Computes the hyperbolic tangent of `x`. The tangent is defined to be the hyperbolic sine of x divided by the hyperbolic cosine of x.

Works on radians only. The no-data value `null` is passed through and therefore gets propagated.

Parameters

x*

An angle in radians.

> Data type: **number, null**

Return Value

The computed hyperbolic tangent of `x`.

> Data type: **number, null**

Examples

Example #1

```
tanh(x = 0) => 0
```

See Also

- Hyperbolic tangent explained by Wolfram MathWorld

text_begins

Text begins with another text

TEXTS COMPARISON

Download JSON

Description

```
text_begins(string|null data, string pattern, ?boolean case_sensitive = true) : boolean|null
```

Checks whether the text (also known as *string*) specified for `data` contains the text specified for `pattern` at the beginning. Both are expected to be encoded in UTF-8 by default. The no-data value `null` is passed through and therefore gets propagated.

Parameters

`data`*

Text in which to find something at the beginning.

> Data type: **string, null**

`pattern`*

Text to find at the beginning of `data`. Regular expressions are not supported.

> Data type: **string**

`case_sensitive = true`

Case sensitive comparison can be disabled by setting this parameter to `false`.

> Data type: **boolean**

Return Value

`true` if `data` begins with `pattern`, false` otherwise.

> Data type: **boolean, null**

Examples

Example #1

```
text_begins(data = "Lorem ipsum dolor sit amet", pattern = "amet") => false
```

Example #2

```
text_begins(data = "Lorem ipsum dolor sit amet", pattern = "Lorem") => true
```

Example #3

```
text_begins(data = "Lorem ipsum dolor sit amet", pattern = "lorem") => false
```

Example #4

```
text_begins(data = "Lorem ipsum dolor sit amet", pattern = "lorem", case_sensitive =
false) => true
```

Example #5

```
text_begins(data = "Ä", pattern = "ä", case_sensitive = false) => true
```

Example #6

```
text_begins(data = null, pattern = "null") => null
```

text_contains

Text contains another text

TEXTS COMPARISON

Download JSON

Description

```
text_contains(string|null data, string pattern, ?boolean case_sensitive = true) :
boolean|null
```

Checks whether the text (also known as *string*) specified for `data` contains the text specified for `pattern`. Both are expected to be encoded in UTF-8 by default. The no-data value `null` is passed through and therefore gets propagated.

Parameters

`data`*

Text in which to find something in.

Data type: **string, null**

`pattern`*

Text to find in `data`. Regular expressions are not supported.

> Data type: **string**

`case_sensitive = true`

Case sensitive comparison can be disabled by setting this parameter to `false`.

> Data type: **boolean**

Return Value

`true` if `data` contains the `pattern`, false` otherwise.

> Data type: **boolean, null**

Examples

Example #1

```
text_contains(data = "Lorem ipsum dolor sit amet", pattern = "openEO") => false
```

Example #2

```
text_contains(data = "Lorem ipsum dolor sit amet", pattern = "ipsum dolor") => true
```

Example #3

```
text_contains(data = "Lorem ipsum dolor sit amet", pattern = "Ipsum Dolor") => false
```

Example #4

```
text_contains(data = "Lorem ipsum dolor sit amet", pattern = "SIT", case_sensitive =
false) => true
```

Example #5

```
text_contains(data = "ÄÖÜ", pattern = "ö", case_sensitive = false) => true
```

Example #6

```
text_contains(data = null, pattern = "null") => null
```

text_ends

Text ends with another text

TEXTS COMPARISON

Download JSON

Description

```
text_ends(string|null data, string pattern, ?boolean case_sensitive = true) : boolean|null
```

Checks whether the text (also known as *string*) specified for `data` contains the text specified for `pattern` at the end. Both are expected to be encoded in UTF-8 by default. The no-data value `null` is passed through and therefore gets propagated.

Parameters

`data`*

Text in which to find something at the end.

> Data type: **string, null**

`pattern`*

Text to find at the end of `data`. Regular expressions are not supported.

> Data type: **string**

`case_sensitive = true`

Case sensitive comparison can be disabled by setting this parameter to `false`.

> Data type: **boolean**

Return Value

`true` if `data` ends with `pattern`, false` otherwise.

> Data type: **boolean, null**

Examples

Example #1

```
text_ends(data = "Lorem ipsum dolor sit amet", pattern = "amet") => true
```

Example #2

```
text_ends(data = "Lorem ipsum dolor sit amet", pattern = "AMET") => false
```

Example #3

```
text_ends(data = "Lorem ipsum dolor sit amet", pattern = "Lorem") => false
```

Example #4

```
text_ends(data = "Lorem ipsum dolor sit amet", pattern = "AMET", case_sensitive =
false) => true
```

Example #5

```
text_ends(data = "Ä", pattern = "ä", case_sensitive = false) => true
```

Example #6

```
text_ends(data = null, pattern = "null") => null
```

text_merge

Concatenate elements to a single text

TEXTS

Download JSON

Description

```
text_merge(array<string|number|boolean|null> data, ?string|number|boolean|null separator =
"") : string
```

Merges text representations (also known as *string*) of a set of elements to a single text, having the separator between each element.

Parameters

data*

A set of elements. Numbers, boolean values and null values get converted to their (lower case) string representation. For example: `1` (integer), `-1.5` (number), `true` / `false` (boolean values)

> Data type: **array<string|number|boolean|null>**

separator = " "

A separator to put between each of the individual texts. Defaults to an empty string.

> Data type: **string, number, boolean, null**

Return Value

A string containing a string representation of all the array elements in the same order, with the separator between each element.

> Data type: **string**

Examples

Example #1

```
text_merge(data = ["Hello","World"], separator = " ") => "Hello World"
```

Example #2

```
text_merge(data = [1,2,3,4,5,6,7,8,9,0]) => "1234567890"
```

Example #3

```
text_merge(data = [null,true,false,1,-1.5,"ß"], separator = "\n") =>
"null\ntrue\nfalse\n1\n-1.5\nß"
```

Example #4

```
text_merge(data = [2,0], separator = 1) => "210"
```

Example #5

```
text_merge(data = []) => ""
```

trim_cube

Remove dimension labels with no-data values

CUBES

Download JSON

Description

`trim_cube(raster-cube data) : raster-cube`

Removes dimension labels solely containing no-data values. If the dimension is irregular categorical then dimension labels in the middle can be removed.

Parameters

data*

A raster data cube to trim.

Data type: **raster-cube**

Return Value

A trimmed raster data cube with the same dimensions. The dimension properties name, type, reference system and resolution remain unchanged. The number of dimension labels may decrease.

> Data type: **raster-cube**

variance

Variance

MATH > STATISTICS REDUCER

Download JSON

Description

```
variance(array<number|null> data, ?boolean ignore_nodata = true) : number|null
```

Computes the sample variance of an array of numbers by calculating the square of the standard deviation (see `sd`). It is defined to be the expectation of the squared deviation of a random variable from its expected value. Basically, it measures how far the numbers in the array are spread out from their average value.

An array without non-`null` elements resolves always with `null`.

Parameters

`data`*

An array of numbers.

> Data type: **array<number|null>**

`ignore_nodata = true`

Indicates whether no-data values are ignored or not. Ignores them by default. Setting this flag to `false` considers no-data values so that `null` is returned if any value is such a value.

> Data type: **boolean**

Return Value

The computed sample variance.

> Data type: **number, null**

Examples

Example #1

```
variance(data = [-1,1,3]) => 4
```

Example #2

```
variance(data = [2,3,3,null,4,4,5]) => 1.1
```

Example #3

```
variance(data = [-1,1,null,3], ignore_nodata = false) => null
```

Example #4

The input array is empty: return `null`.

```
variance(data = []) => null
```

See Also

- Variance explained by Wolfram MathWorld

xor

Logical XOR (exclusive or)

LOGIC

Download JSON

Description

`xor(boolean|null x, boolean|null y) : boolean|null`

Checks if **exactly one** of the values is true. If a component is `null`, the result will be `null` if the outcome is ambiguous.

Truth table:

```
a \ b || null | false | true
----- || ---- | ----- | -----
null  || null | null  | null
false || null | false | true
true  || null | true  | false
```

Parameters

x*

A boolean value.

Data type: **boolean, null**

y*

A boolean value.

> Data type: **boolean, null**

Return Value

Boolean result of the logical XOR.

> Data type: **boolean, null**

Examples

Example #1

```
xor(x = true, y = true) => false
```

Example #2

```
xor(x = false, y = false) => false
```

Example #3

```
xor(x = true, y = false) => true
```

Example #4

```
xor(x = true, y = null) => null
```

Example #5

```
xor(x = false, y = null) => null
```