# eodatasets3

Geoscience Australia

Nov 24, 2021

# CONTENTS

EO Datasets aims to be the easiest way to write, validate and convert dataset imagery and metadata for the Open Data Cube

There are two major tools for creating datasets:

1. *DatasetAssembler*, for writing a package: including writing the metadata document, COG imagery, thumbnails, checksum files etc.
2. *DatasetPrepare*, for preparing a metadata document, referencing existing imagery and files.

Their APIs are the same, except the assembler adds methods named `write_*` for writing new files.

---

**Note:** methods named `note_*` will note an existing file in the metadata, while `write_*` will write the file into the package too.

---

CHAPTER

# ONE

# ASSEMBLE A DATASET PACKAGE

Here's a simple example of creating a dataset package with one measurement (called "blue") from an existing image.

The measurement is converted to a COG image when written to the package:

```
from eodatasets3 import DatasetAssembler

with DatasetAssembler(collection, naming_conventions='default') as p:
    p.product_family = "blues"

    # Date of acquisition (UTC if no timezone).
    p.datetime = datetime(2019, 7, 4, 13, 7, 5)
    # When the data was processed/created.
    p.processed_now() # Right now!
    # (If not newly created, set the date on the field: `p.processed = ...`)

    # Write our measurement from the given path, calling it 'blue'.
    p.write_measurement("blue", blue_geotiff_path)

    # Add a jpg thumbnail using our only measurement for the r/g/b bands.
    p.write_thumbnail("blue", "blue", "blue")

    # Complete the dataset.
    p.done()
```

**Note:** Note that until you call `done()`, nothing will exist in the dataset's final output location. It is stored in a hidden temporary folder in the output directory, and renamed by `done()` once complete and valid.

CHAPTER

TWO

# WRITING ONLY A METADATA DOC

(ie. “I already have appropriate imagery files!”)

Example of generating a metadata document with *DatasetPrepare*:

```
collection_path = integration_data_path
```

```
from eodatasets3 import DatasetPrepare

usgs_level1 = collection_path / 'LC08_L1TP_090084_20160121_20170405_01_T1'
metadata_path = usgs_level1 / 'odc-metadata.yaml'

with DatasetPrepare(
    metadata_path=metadata_path,
) as p:
    p.product_family = "level1"
    p.datetime = datetime(2019, 7, 4, 13, 7, 5)
    p.processed_now()

    # Note the measurement in the metadata. (instead of ``write``)
    p.note_measurement('red',
       usgs_level1 / 'LC08_L1TP_090084_20160121_20170405_01_T1_B4.TIF'
    )

    # Or give the path relative to the dataset location
    # (eg. This will work unchanged on non-filesystem locations, such as ``s3://`` or tar␣
↪files)
    p.note_measurement('blue',
       'LC08_L1TP_090084_20160121_20170405_01_T1_B2.TIF',
       relative_to_dataset_location=True
    )

    # Add links to other files included in the package ("accessories"), such as
    # alternative metadata files.
    [mtl_path] = usgs_level1.glob('*_MTL.txt')
    p.note_accessory_file('metadata:mtl', mtl_path)

    # Add whatever else you want.
    ...

    # Validate and write our metadata document!
    p.done()
```

(continues on next page)

(continued from previous page)

```
# We created a metadata file!
assert metadata_path.exists()
```

Custom properties can also be set directly on `.properties`:

```
p.properties['fmask:cloud_cover'] = 34.0
```

And known properties are automatically normalised:

```
p.platform = "LANDSAT_8"  # to: 'landsat-8'
p.processed = "2016-03-04 14:23:30Z"  # into a date.
p.maturity = "FINAL"  # lowercased
p.properties["eo:off_nadir"] = "34"  # into a number
```

---

**Note:** By default, a validation error will be thrown if a referenced file lives outside the dataset (location), as this will require absolute paths. Relative paths are considered best-practice for Open Data Cube metadata.

You can allow absolute paths in your metadata document using a field on construction (*DatasetPrepare()*):

```
with DatasetPrepare(
   dataset_location=usgs_level1,
   allow_absolute_paths=True,
 ):
     ...
```

---

CHAPTER

# THREE

# INCLUDING PROVENANCE

Most datasets are processed from an existing input dataset and have the same spatial information as the input. We can record them as source datasets, and the assembler can optionally copy any common metadata automatically:

```
collection = Path('/some/output/collection/path')
with DatasetAssembler(collection) as p:
    # We add a source dataset, asking to inherit the common properties
    # (eg. platform, instrument, datetime)
    p.add_source_path(level1_ls8_dataset_path, auto_inherit_properties=True)

    # Set our product information.
    # It's a GA product of "numerus-unus" ("the number one").
    p.producer = "ga.gov.au"
    p.product_family = "numerus-unus"
    p.dataset_version = "3.0.0"

    ...
```

In these situations, we often write our new pixels as a numpy array, inheriting the existing *grid spatial information* of our input dataset:

```
# Write a measurement from a numpy array, using the source dataset's grid spec.
p.write_measurement_numpy(
    "water",
    my_computed_numpy_array,
    GridSpec.from_dataset_doc(source_dataset),
    nodata=-999,
)
```

Other ways to reference your source datasets:

- As an in-memory *DatasetDoc* using *p.add_source_dataset()*
- Or as raw uuids, using *p.note_source_datasets()* (without property inheritance)

CHAPTER

# FOUR

# CREATING DOCUMENTS IN-MEMORY

You may want to assemble metadata entirely in memory without touching the filesystem.

To do this, prepare a dataset as normal. You still need to give a dataset location, as paths in the document will be relative to this location:

```
>>> from eodatasets3 import DatasetPrepare
>>>
>>> p = DatasetPrepare(dataset_location=dataset_location)
>>> p.datetime = datetime(2019, 7, 4, 13, 7, 5)
>>> p.product_name = "loch_ness_sightings"
>>> p.processed = datetime(2019, 7, 4, 13, 8, 7)
```

Normally when a measurement is added, the image will be opened to read grid and size informaation. You can avoid this by giving a *GridSpec* yourself (see *GridSpec doc* for creation):

```
>>> p.note_measurement(
...     "blue",
...     measurement_path,
...     # We give it grid information, so it doesn't have to read it itself.
...     grid=grid_spec,
...     # And the image pixels, since we are letting it calculate our geometry.
...     pixels=numpy.ones((60, 60), numpy.int16),
...     nodata=-1,
... )
```

---

**Note:** If you're writing your own image files manually, you may still want to use eodataset's name generation. You can ask for suitable paths from *p.names*:

See the *the naming section* for examples.

---

Now finish it as a *DatasetDoc*:

```
>>> dataset = p.to_dataset_doc()
```

You can now use *serialise functions* on the result yourself, such as conversion to a dictionary:

```
>>> from eodatasets3 import serialise
>>> doc: dict = serialise.to_doc(dataset)
>>> doc['label']
'loch_ness_sightings_2019-07-04'
```

Or convert it to a formatted yaml: *serialise.to_path(path, dataset)* or *serialise.to_stream(stream, dataset)*.

CHAPTER

FIVE

# AVOIDING GEOMETRY CALCULATION

Datasets include a geometry field, which shows the coverage of valid data pixels of all measurements.

By default, the assembler will create this geometry by reading the pixels from your measurements, and calculate a geometry vector on completion.

This can be configured by setting the *p.valid_data_method* to a different *ValidDataMethod* value.

But you may want to avoid these reads and calculations entirely, in which case you can set a geometry yourself:

```
p.geometry = my_shapely_polygon
```

Or copy it from one of your source datasets when you add your provenance (if it has the same coverage):

```
p.add_source_path(source_path, inherit_geometry=True)
```

If you do this before you *note* measurements, it will not need to read any pixels from them.

CHAPTER

SIX

# GENERATING NAMES & PATHS ALONE

You can use the naming module alone to find file paths:

```
import eodatasets3
from pathlib import Path
from eodatasets3 import DatasetDoc
```

Create some properties.

```
d = DatasetDoc()
d.platform = "sentinel-2a"
d.product_family = "fires"
d.datetime = "2018-05-04T12:23:32"
d.processed_now()

# Arbitrarily set any properties.
d.properties["fmask:cloud_shadow"] = 42.0
d.properties.update({"odc:file_format": "GeoTIFF"})
```

**Note:** You can use a plain dict if you prefer. But we use an *DatasetDoc()* here, which has convenience methods similar to *DatasetAssembler* for building properties.

Now create a *namer* instance with our properties (and chosen naming conventions):

```
names = eodatasets3.namer(d, conventions="default")
```

We can see some generated names:

```
print(names.metadata_file)
print(names.measurement_filename('water'))
print()
print(names.product_name)
print(names.dataset_folder)
```

Output:

```
s2a_fires_2018-05-04.odc-metadata.yaml
s2a_fires_2018-05-04_water.tif

s2a_fires
s2a_fires/2018/05/04
```

In reality, these paths go within a location (folder, s3 bucket, etc) somewhere.

This location is called the *collection_prefix*, and we can create our namer with one:

```
collection_path = Path('/datacube/collections')

names = eodatasets3.namer(d, collection_prefix=collection_path)

print("The dataset location is always a URL:")
print(names.dataset_location)

print()

a_file_name = names.measurement_filename('water')
print(f"We can resolve our previous file name to a dataset URL:")
print(names.resolve_file(a_file_name))

print()

print(f"Or a local path (if it's file://):")
print(repr(names.resolve_path(a_file_name)))
```

```
The dataset location is always a URL:
file:///datacube/collections/s2a_fires/2018/05/04/

We can resolve our previous file name to a dataset URL:
file:///datacube/collections/s2a_fires/2018/05/04/s2a_fires_2018-05-04_water.tif

Or a local path (if it's file://):
PosixPath('/datacube/collections/s2a_fires/2018/05/04/s2a_fires_2018-05-04_water.tif')
```

**Note:** The collection prefix can also be a remote url: `https://example.com/collections`, `s3://  ...` … etc

We could now start assembling some metadata if our dataset doesn't exist, passing it our existing fields:

```
# Our dataset doesn't exist?
if not names.dataset_path.exists():
    with DatasetAssembler(names=names) as p:

        # The properties are already set, thanks to our namer.

        ... # Write some measurements here, etc!

        p.done()

# It exists!
assert names.dataset_path.exists()
```

**Note:** `.dataset_path` is a convenience property to get the `.dataset_location` as a local Path, if possible.

**Note:** The assembler classes don't yet support writing to remote locations! But you can use the above api to write it

yourself manually (for now).

CHAPTER

SEVEN

# NAMING THINGS YOURSELF

Names and paths are only auto-generated if they have not been set manually by the user.

You can set properties yourself on the *NamingConventions* to avoid automatic generation (or to avoid their finicky metadata requirements).

```
>>> p = DatasetPrepare(collection_path)
>>> p.platform = 'sentinel-2a'
>>> p.product_family = 'ard'
>>> # The namer will generate a product name:
>>> p.names.product_name
's2a_ard'
>>> # Let's customise the generated abbreviation:
>>> p.names.platform_abbreviated = "s2"
>>> p.names.product_name
's2_ard'
```

See more examples in the assembler *.names* property.

CHAPTER

# EIGHT

# SEPARATING METADATA FROM DATA

(Or. "I don't want to store ODC metadata alongside my data!")

You may want your data to live in a different location to your ODC metadata files, or even not store metadata on disk at all. But you still want it to be easily indexed.

To do this, the `done()` commands include an `embed_location=True` argument. This will tell the Assemblers to embed the `dataset_location` into the output document.

**Note:** When indexing a dataset, if ODC finds an embedded location, it uses it in place of the metadata document's own location.

For example:

```
metadata_path = tmp_path / "my-dataset.odc-metadata.yaml"

with DatasetPrepare(
    # Our collection location is different to our metadata location!
    collection_location="s3://dea-public-data-dev",
    metadata_path=metadata_path,
    allow_absolute_paths=True,
) as p:
    p.dataset_id = UUID("8c9e907a-df35-407c-a89b-920d0b24fdbf")
    p.datetime = datetime(2019, 7, 4, 13, 7, 5)
    p.product_family = "quaternarius"
    p.processed_now()

    p.note_measurement("blue", blue_geotiff_path)

    # When writing, embed our dataset location in the output
    p.done(embed_location=True)

# Print the header of the document:
output =  metadata_path.read_text()
print(output[:output.find('crs:')].strip())
```

Now our dataset location is included in the document:

```
---
# Dataset
$schema: https://schemas.opendatacube.org/dataset
id: 8c9e907a-df35-407c-a89b-920d0b24fdbf
```

(continues on next page)

(continued from previous page)

```
label: quaternarius_2019-07-04
product:
  name: quaternarius

location: s3://dea-public-data-dev/quaternarius/2019/07/04/
```

Now ODC will ignore the actual location of the metadata file we are indexing, and use the embedded `s3` location instead.

---

**Note:** Note that we added `allow_absolute_paths=True` for our own testing simplicity in this guide.

In reality, your measurements should live in that same `s3://` location, and so they'll end up relative.

---

CHAPTER

NINE

# UNDERSTANDING LOCATIONS

When writing an ODC dataset, there are two important locations that need to be known by assembler: where the metadata file will go, and the "dataset location".

**Note:** In ODC, all file paths in a dataset are computed relative to the dataset_location

Examples

| Dataset Location | Path | Result |
|---|---|---|
| s3://dea-public-data/year-summary/2010/ | water.tif | s3://dea-public-data/year-summary/2010/water.tif |
| s3://dea-public-data/year-summary/2010/ | bands/water.tif | s3://dea-public-data/year-summary/2010/bands/water.tif |
| file:///rs0/datacube/LS7_NBAR/10_-24/odc-metadata.yaml | v1496652530.nc | file:///rs0/datacube/LS7_NBAR/10_-24/v1496652530.nc |
| file:///rs0/datacube/LS7_NBAR/10_-24/odc-metadata.yaml | s3://dea-public-data/year-summary/2010/water.tif | s3://dea-public-data/year-summary/2010/water.tif |

You can specify both of these paths if you want:

```
with DatasetPrepare(dataset_location=..., metadata_path=...):
    ...
```

But you usually don't need to give them explicitly. They will be inferred if missing.

1. If you only give a metadata path, the dataset location will be the same.:

```
metadata_path             = "file:///tmp/ls7_nbar_20120403_c1/my-dataset.odc-
↪metadata.yaml"
inferred_dataset_location = "file:///tmp/ls7_nbar_20120403_c1/my-dataset.odc-
↪metadata.yaml"
```

2. If you only give a dataset location, a metadata path will be created as a sibling file with `.odc-metadata.yaml` suffix within the same "directory" as the location.:

```
dataset_location       = "file:///tmp/ls7_nbar_20120403_c1/my-dataset.tar"
inferred_metadata_path = "file:///tmp/ls7_nbar_20120403_c1/my-dataset.odc-metadata.
↪yaml"
```

3. ... or you can give neither of them! And they will be generated from a base *collection_path*.:

```
collection_path           = "file:///collections"
inferred_dataset_location = "file:///collections/ga_s2am_level4_3/023/543/2013/02/
↪03/
inferred_metadata_path    = "file:///collections/ga_s2am_level4_3/023/543/2013/02/
↪03/ga_s2am_level4_3-2-3_023543_2013-02-03_final.odc-metadata.yaml
```

**Note:** For local files, you can give the location as a `pathlib.Path`, and it will internally be converted into a URL for you.

In the third case, the folder and file names are generated from your metadata properties and chosen naming convention. You can also *set folders, files and parts yourself*.

Specifying a collection path:

```
with DatasetPrepare(collection_path=collection, naming_conventions='default'):
    ...
```

Let's print out a table of example default paths for each built-in naming convention:

```
from eodatasets3 import namer, DatasetDoc
import eodatasets3.names

# Build an example dataset
p = DatasetDoc()
p.platform = "sentinel-2a"
p.instrument = "MSI"
p.datetime = datetime(2013, 2, 3, 6, 5, 2)
p.region_code = "023543"
p.processed_now()
p.producer = "ga.gov.au"
p.dataset_version = "1.2.3"
p.product_family = "level4"
p.maturity = 'final'
p.collection_number = 3
p.properties['sentinel:sentinel_tile_id'] = 'S2B_OPER_MSI_L1C_TL_VGS4_20210426T010904_
↪A021606_T56JMQ_N03.00'

collection_prefix = 'https://test-collection'

# Print the result for each known convention
header = f"{'convention':20} {'metadata_file':64} dataset_location"
print(header)
print('-' * len(header))
for conventions in eodatasets3.names.KNOWN_CONVENTIONS.keys():
    n = namer(p, conventions=conventions, collection_prefix=collection_prefix)
    print(f"{conventions:20} {str(n.metadata_file):64} {n.dataset_location}")
```

Result:

```
convention           metadata_file                                                ␣
↪dataset_location
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------------------
↪-------------
default              ga_s2am_level4_3-2-3_023543_2013-02-03_final.odc-metadata.yaml  ␣
↪https://test-collection/ga_s2am_level4_3/023/543/2013/02/03/
dea                  ga_s2am_level4_3-2-3_023543_2013-02-03_final.odc-metadata.yaml  ␣
↪https://test-collection/ga_s2am_level4_3/023/543/2013/02/03/
dea_s2               ga_s2am_level4_3-2-3_023543_2013-02-03_final.odc-metadata.yaml  ␣
↪https://test-collection/ga_s2am_level4_3/023/543/2013/02/03/20210426T010904/
dea_s2_derivative    ga_s2_level4_3_023543_2013-02-03_final.odc-metadata.yaml        ␣
↪https://test-collection/ga_s2_level4_3/1-2-3/023/543/2013/02/03/20210426T010904/
dea_c3               ga_s2_level4_3_023543_2013-02-03_final.odc-metadata.yaml        ␣
↪https://test-collection/ga_s2_level4_3/1-2-3/023/543/2013/02/03/
deafrica             level4_s2_023543_2013-02-03_final.odc-metadata.yaml             ␣
↪https://test-collection/level4_s2/1-2-3/023/543/2013/02/03/
```

**Note:** The `default` conventions look the same as `dea` here, but `dea` is stricter in its mandatory metadata fields (following policies within the organisation).

You can leave out many more properties from your metadata in `default` and they will not be included in the generated paths.

CHAPTER

# TEN

# DATASET PREPARE CLASS REFERENCE

class eodatasets3.**DatasetPrepare**(*collection_location=None*, *, *dataset_location=None*, *metadata_path=None*, *dataset_id=None*, *allow_absolute_paths=False*, *naming_conventions=None*, *names=None*, *dataset=None*)

Prepare dataset metadata

**INHERITABLE_PROPERTIES = {'datetime', 'dtr:end_datetime', 'dtr:start_datetime', 'eo:cloud_cover', 'eo:constellation', 'eo:gsd', 'eo:instrument', 'eo:platform', 'eo:sun_azimuth', 'eo:sun_elevation', 'fmask:clear', 'fmask:cloud', 'fmask:cloud_shadow', 'fmask:snow', 'fmask:water', 'gqa:abs_iterative_mean_x', 'gqa:abs_iterative_mean_xy', 'gqa:abs_iterative_mean_y', 'gqa:abs_x', 'gqa:abs_xy', 'gqa:abs_y', 'gqa:cep90', 'gqa:iterative_mean_x', 'gqa:iterative_mean_xy', 'gqa:iterative_mean_y', 'gqa:iterative_stddev_x', 'gqa:iterative_stddev_xy', 'gqa:iterative_stddev_y', 'gqa:mean_x', 'gqa:mean_xy', 'gqa:mean_y', 'gqa:stddev_x', 'gqa:stddev_xy', 'gqa:stddev_y', 'landsat:collection_category', 'landsat:collection_number', 'landsat:landsat_product_id', 'landsat:landsat_scene_id', 'landsat:rmse', 'landsat:rmse_x', 'landsat:rmse_y', 'landsat:scene_id', 'landsat:wrs_path', 'landsat:wrs_row', 'mission', 'odc:region_code', 'sat:absolute_orbit', 'sat:anx_datetime', 'sat:orbit_state', 'sat:platform_international_designator', 'sat:relative_orbit', 'sentinel:datastrip_id', 'sentinel:datatake_start_datetime', 'sentinel:grid_square', 'sentinel:latitude_band', 'sentinel:sentinel_tile_id', 'sentinel:utm_zone'}**

The properties that will automatically be inherited from a source dataset when *auto_inherit_properties=True*

These are fields that are inherent to the underlying observation, and so will still be relevant after most 1:1 processing.

**__init__**(*collection_location=None*, *, *dataset_location=None*, *metadata_path=None*, *dataset_id=None*, *allow_absolute_paths=False*, *naming_conventions=None*, *names=None*, *dataset=None*)

Build an EO3 metadata document, with functions for reading information from imagery and calculating names and paths.

In addition to the below documented methods, metadata fields can read and set using *Eo3Interface's* fields.

There are three optional paths that can be specified. At least one must be specified. Collection, dataset or metadata path.

- A `collection_path` is the root folder where datasets will live (in sub-[sub]-folders).
- Each dataset has its own `dataset_location`, as stored in an Open Data Cube index. All paths inside the metadata document are relative to this location.
- An output `metadata_path` document location*.

If you're writing data, you typically only need to specify the collection path, and the others will be automatically generated using the naming conventions.

If you're only writing a metadata file (for existing data), you only need to specify a metadata path.

If you're storing data using an exotic URI schema, such as a 'tar:///' URL path, you will need to specify this as your dataset location.

**Parameters**

- **collection_location** (Union[str, Path, None]) – Optional base directory where the collection of datasets should live. Subfolders will be created accordion to the naming convention.
- **dataset_location** (Union[str, Path, None]) – Optional location for this dataset.

  Otherwise it will be generated according to the collection path and naming conventions.

  (this is as indexed into ODC – ie. a file name).
- **metadata_path** (Union[str, Path, None]) – Optional metadata document output path. Otherwise it will be generated according to the collection path and naming conventions.
- **dataset_id** (Optional[UUID]) – Optional UUID for this dataset, otherwise a random one will be created. Use this if you have a stable way of generating your own IDs.
- **allow_absolute_paths** (bool) – Allow metadata paths to refer to files outside the dataset location. this means they will have to be absolute paths, and not be portable. (default: False)
- **naming_conventions** (Optional[str]) – Naming conventions to use. Supports *default* or *dea*. The latter has stricter metadata requirements (try it and see – it will tell your what's missing).

**add_source_dataset**(*dataset*, *classifier=None*, *auto_inherit_properties=False*, *inherit_geometry=False*, *inherit_skip_properties=None*)

Record a source dataset using its metadata document.

It can optionally copy common properties from the source dataset (platform, instrument etc)/

(see *INHERITABLE_PROPERTIES* for the list of fields that are inheritable)

**Parameters**

- **dataset** (*DatasetDoc*) –
- **auto_inherit_properties** (bool) – Whether to copy any common properties from the dataset
- **classifier** (Optional[str]) – How to classify the kind of source dataset. This is will automatically be filled with the family of dataset if available (eg. "level1").

  You want to set this if you have two datasets of the same type that are used for different purposes. Such as having a second level1 dataset that was used for QA (but is not this same scene).
- **inherit_geometry** (bool) – Instead of re-calculating the valid bounds geometry based on the data, which can be very computationally expensive e.g. Landsat 7 striped data, use the valid data geometry from this source dataset.
- **inherit_skip_properties** (Optional[str]) – An extra list of property names that should not be copied. This is useful when generating summaries which combine multiple input source datasets.

See *add_source_path()* if you have a filepath reference instead of a document.

**add_source_path**(**paths*, *classifier=None*, *auto_inherit_properties=False*, *inherit_geometry=False*)
Record a source dataset using the path to its metadata document.

**Parameters**

- **paths** (Path) – Filesystem path(s) to source metadata documents
- **classifier** (Optional[str]) – How to classify the kind of source dataset. This is will automatically be filled with the family of dataset if available (eg. "level1").

  You want to set this if you have two datasets of the same type that are used for different purposes. Such as having a second level1 dataset that was used for QA (but is not this same scene).
- **auto_inherit_properties** (bool) – Whether to copy any common properties from the dataset
- **inherit_geometry** (bool) – Instead of re-calculating the valid bounds geometry based on the data, which can be very computationally expensive e.g. Landsat 7 striped data, use the valid data geometry from this source dataset.

See also *add_source_dataset()*

**done**(*validate_correctness=True*, *sort_measurements=True*, *embed_location=False*)
Write the prepared metadata document to the given output path.

**Return type** Tuple[UUID, Path]

**geometry: Optional[shapely.geometry.base.BaseGeometry]**
Valid-data polygon, in the same CRS as the measurements.

This must cover all valid pixels to be valid in ODC (it's allowed to be larger than the valid pixel area, but not smaller).

It will be computed automatically from measurements if not set manually. You can also inherit it from source datasets in the `add_source_*()` methods.

**iter_measurement_paths**()

> **Warning:** *not recommended* for use - will likely change soon.

Iterate through the list of measurement names that have been written, and their current (temporary) paths.

TODO: Perhaps we want to return a real measurement structure here as it's not very extensible.

**Return type** Generator[Tuple[*GridSpec*, str, Path], None, None]

**property label: Optional[str]**
An optional displayable string to identify this dataset.

These are often used when when presenting a list of datasets, such as in search results or a filesystem folder. They are unstructured, but should be more humane than showing a list of UUIDs.

By convention they have no spaces, due to their usage in filenames.

Eg. `ga_ls5t_ard_3-0-0_092084_2009-12-17_final` or USGS's `LT05_L1TP_092084_20091217_20161017_01_T1`

A label will be auto-generated using the naming-conventions, but you can manually override it by setting this property.

**Return type** Optional[str]

**names:** *eodatasets3.names.NamingConventions*

The name generator (an instance of *NamingConventions*)

By default, all names will be generated based on metadata fields and the chosen naming conventions.

But you can set your own names here manually to avoid the magic.

(for the devious among you, this can also avoid metadata field requirements for name generation).

Examples:

Set a product name:

```
p.names.product_name = 'my_product_name'
```

Manually set the abbreviations used in name generation

(By default, for example, landsat-7 will be abbreviated to "ls7". But maybe you want "ls" in all your datasets):

```
p.names.platform_abbreviated = "ls"
# Other abbreviations:
p.names.instrument_abbreviated = "e"
p.names.producer_abbreviated = "usgs"
```

Set your own label (the human identifier for the dataset, and the default prefix of filenames):

```
p.names.dataset_label = "landsat-observations-12th-may-2021"
```

Customise the dataset's folder offset:

```
>>> p.names.dataset_folder
'ga_ls8c_ones_3/090/084/2016/01/21'
```

... to use a custom time hierarchy:

```
>>> p.names.time_folder = p.datetime.strftime("years/%Y")
>>> p.names.dataset_folder
'ga_ls8c_ones_3/090/084/years/2016'
```

... or a custom region format:

```
>>> p.names.region_folder = 'x04y23'
>>> p.names.dataset_folder
'ga_ls8c_ones_3/x04y23/years/2016'
```

... or replace it altogether:

```
p.names.dataset_folder = "datasets/january/2021"
```

Configure the pattern used for generating filenames:

```
p.names.filename_pattern = "my-file.{file_id}.{suffix}"
```

---

**Note:** All filenames are given a `{file_id}` (eg. `"odc-metadata"` or `""`) and `{suffix}` (eg. `"yaml"`) variable to distinguish themselves.

(Patterns can also contain folder separators. It will be relative to the dataset folder)

The path to the EO3 metadata doc (relative path to the dataset location):

```
p.names.metadata_file = "my-metadata.odc-metadata.yaml"
```

The URI for the product:

```
p.names.product_uri = "https://collections.earth.test.example/product/my-product
↪"
```

A full list of fields can be seen on *eodatasets3.NamingConventions*

**note_accessory_file**(*name*, *path*)

Record a reference to an additional file that's included in the dataset, but is not a band/measurement.

Such as non-ODC metadata, thumbnails, checksums, etc. Any included file that is not recorded in the measurements.

By convention, the name should have prefixes with their category, such as `metadata:` or `thumbnail:`.

eg. `metadata:landsat_processor`, `checksum:sha1`, `thumbnail:full`.

**Parameters**

- **name** (str) – identifying name, eg `metadata:mtl`
- **path** (Union[Path, str]) – local path to file.

**note_measurement**(*name*, *path*, *expand_valid_data=True*, *relative_to_dataset_location=False*, *grid=None*, *pixels=None*, *nodata=None*)

Reference a measurement from its existing path. It may be a Path or any URL resolvable by rasterio.

By default, a relative path is relative to your current directory. You may want to specify `relative_to_dataset_location=True`.

The path will be opened to read geo and pixel information, unless you specify the information yourself (grid, pixels, nodata). (the latter two only needed if expand_valid_data==True)

**Parameters**

- **name** – measurement name
- **path** (Union[Path, str]) – path to measurement
- **expand_valid_data** – Expand the valid data bounds with this measurement's valid data.
- **relative_to_dataset_location** – Should this be read relative to the dataset location? (requires a computed dataset location)

**note_source_datasets**(*classifier*, **dataset_ids*)

Expand the lineage with raw source dataset ids.

**Note:** If you have direct access to the datasets, you probably want to use *add_source_path()* or *add_source_dataset()*, so that fields can be inherited from them automatically.

**Parameters**

- **classifier** (str) – How to classify the source dataset.

By convention, this is usually the family of the source dataset (properties->odc:product_family). Such as "level1".

A classifier is used to distinguish source datasets of the same product that are used differently.

Such as a normal source level1 dataset (classifier: "level1"), and a second source level1 that was used only for QA (classifier: "qa").

- **dataset_ids** (Union[str, UUID]) – The UUIDs of the source datasets

**note_thumbnail**(*thumb_path*, *kind=None*)
Record a reference to a thumbnail path.

Optionally specify the "kind" of thumbnail if there are multiple to distinguish between. eg. 'full'

**to_dataset_doc**(*dataset_location=None*, *embed_location=False*, *validate_correctness=True*, *sort_measurements=True*, *expect_geometry=True*)
Create the metadata doc as an in-memory *eodatasets3.DatasetDoc* instance.

(You can manually write this out using *serialise.to_path():* or *serialise.to_stream()*)

**Return type** *DatasetDoc*

**valid_data_method:** ***eodatasets3.images.ValidDataMethod***
What method to use to calculate the valid data geometry?

Defaults to *eodatasets3.ValidDataMethod.thorough*

You may change this property before finishing your package.

Eg:

```
p.valid_data_method = ValidDataMethod.filled
```

**write_eo3**(*path=None*, *embed_location=False*, *validate_correctness=True*, *sort_measurements=True*)
Write the prepared metadata document to the given output path.

**Return type** Tuple[UUID, Path]

CHAPTER

# ELEVEN

# DATASET ASSEMBLER CLASS REFERENCE

class eodatasets3.**DatasetAssembler**(*collection_location=None*, ***, *dataset_location=None*, *metadata_path=None*, *dataset_id=None*, *if_exists=IfExists.ThrowError*, *allow_absolute_paths=False*, *naming_conventions='default'*, *names=None*, *dataset=None*)

Assemble a package of a dataset, including metadata, writing COG images, thumbnails, checksums etc.

You may want to use *eodatasets3.DatasetPrepare* if you only need a metadata document.

**__init__**(*collection_location=None*, ***, *dataset_location=None*, *metadata_path=None*, *dataset_id=None*, *if_exists=IfExists.ThrowError*, *allow_absolute_paths=False*, *naming_conventions='default'*, *names=None*, *dataset=None*)

Assemble a dataset with ODC metadata, writing metadata and (optionally) its imagery as COGs.

In addition to the below documented methods, metadata can read and set using *Eo3Interface's* fields.

There are three optional paths that can be specified. At least one must be specified. Collection, dataset or metadata path.

- A `collection_path` is the root folder where datasets will live (in sub-[sub]-folders).
- Each dataset has its own `dataset_location`, as stored in an Open Data Cube index. All paths inside the metadata document are relative to this location.
- An output `metadata_path` document location*.

If you're writing data, you typically only need to specify the collection path, and the others will be automatically generated using the naming conventions.

If you're only writing a metadata file (for existing data), you only need to specify a metadata path.

If you're storing data using an exotic URI schema, such as a 'tar://' URL path, you will need to specify this as your dataset location.

**Parameters**

- **collection_location** (`Optional[Path]`) – Optional base directory where the collection of datasets should live. Subfolders will be created accordion to the naming convention.
- **dataset_location** (`Union[str, Path, None]`) – Optional location for this specific dataset. Otherwise it will be generated according to the collection path and naming conventions.
- **metadata_path** (`Optional[Path]`) – Optional metadata document output path. Otherwise it will be generated according to the collection path and naming conventions.
- **dataset_id** (`Optional[UUID]`) – Optional UUID for this dataset, otherwise a random one will be created. Use this if you have a stable way of generating your own IDs.

- **if_exists** (*IfExists*) – What to do if the output dataset already exists? By default, throw an error.
- **allow_absolute_paths** (bool) – Allow metadata paths to refer to files outside the dataset location. this means they will have to be absolute paths, and not be portable. (default: False)
- **naming_conventions** (str) – Naming conventions to use. Supports *default* or *dea*. The latter has stricter metadata requirements (try it and see – it will tell your what's missing).

**cancel**()
Cancel the package, cleaning up temporary files.

This works like *close()*, but is intentional, so no warning will be raised for forgetting to complete the package first.

**close**()
Clean up any temporary files, even if dataset has not been written

**done**(*validate_correctness=True*, *sort_measurements=True*, *embed_location=False*)
Write the dataset and move it into place.

It will be validated, metadata will be written, and if all is correct, it will be moved to the output location.

The final move is done atomically, so the dataset will only exist in the output location if it is complete.

**Parameters**

- **validate_correctness** (bool) – Run the eo3-validator on the resulting metadata.
- **sort_measurements** (bool) – Order measurements alphabetically. (instead of insert-order)
- **embed_location** (Optional[bool]) – Include the dataset location in the metadata document? When 'None', it will automatically do it if the location is different to metadata doc.

**Raises** *IncompleteDatasetError* If any critical metadata is incomplete.

**Return type** Tuple[UUID, Path]

**Returns** The id and final path to the dataset metadata file.

**extend_user_metadata**(*section_name*, *doc*)
Record extra metadata from the processing of the dataset.

It can be any document suitable for yaml/json serialisation, and will be written into the sidecar "proc-info" metadata.

This is typically used for recording processing parameters or environment information.

**Parameters**

- **section_name** (str) – Should be unique to your product, and identify the kind of document, eg 'brdf_ancillary'
- **doc** (Dict[str, Any]) – Document

**note_accessory_file**(*name*, *path*)
Record a reference to an additional file that's included in the dataset, but is not a band/measurement.

Such as non-ODC metadata, thumbnails, checksums, etc. Any included file that is not recorded in the measurements.

By convention, the name should have prefixes with their category, such as `metadata:` or `thumbnail:`.

eg. `metadata:landsat_processor`, `checksum:sha1`, `thumbnail:full`.

**Parameters**

- **name** (`str`) – identifying name, eg `metadata:mtl`
- **path** (`Union[Path, str]`) – local path to file.

**note_software_version**(*name*, *url*, *version*)

Record the version of some software used to produce the dataset.

**Parameters**

- **name** (`str`) – a short human-readable name for the software. eg "datacube-core"
- **url** (`str`) – A URL where the software is found, such as the git repository.
- **version** (`str`) – the version string, eg. "1.0.0b1"

**write_measurement**(*name*, *input_path*, *index=None*, *overviews=(8, 16, 32)*, *overview_resampling=Resampling.average*, *expand_valid_data=True*, *file_id=None*, *path=None*)

Write a measurement by copying it from a file path.

Assumes the file is gdal-readable.

**Parameters**

- **name** (`str`) – Identifier for the measurement eg `'blue'`.
- **input_path** (`Union[Path, str]`) – The image to read
- **index** (`Optional[int]`) – Which index to read from the image, if it contains more than one.
- **overviews** (`Iterable[int]`) – Set of overview sizes to write
- **overview_resampling** (`Resampling`) – rasterio Resampling method to use
- **expand_valid_data** (`bool`) – Include this measurement in the valid-data geometry of the metadata.
- **file_id** (`Optional[str]`) – Optionally, how to identify this in the filename instead of using the name. (DEA has measurements called `blue`, but their written filenames must be `band04` by convention.)
- **path** (`Optional[Path]`) – Optional path to the image to write. Can be relative to the dataset.

**write_measurement_numpy**(*name*, *array*, *grid_spec*, *nodata=None*, *overviews=(8, 16, 32)*, *overview_resampling=Resampling.average*, *expand_valid_data=True*, *file_id=None*, *path=None*)

Write a measurement from a numpy array and grid spec.

The most common case is to copy the grid spec from your input dataset, assuming you haven't reprojected.

Example:

```
p.write_measurement_numpy(
    "blue",
    new_array,
    GridSpec.from_dataset_doc(source_dataset),
    nodata=-999,
)
```

See *write_measurement()* for other parameters.

**Parameters**

- **array** (ndarray) –
- **grid_spec** (*GridSpec*) –
- **nodata** (Union[int, float, None]) –

**write_measurement_rio**(*name*, *ds*, *index=None*, *overviews=(8, 16, 32)*, *overview_resampling=Resampling.average*, *expand_valid_data=True*, *file_id=None*, *path=None*)

Write a measurement by reading it from an open rasterio dataset

**Parameters**

- **ds** (DatasetReader) – An open rasterio dataset
- **index** (Optional[int]) – Which index to read from the image, if it contains more than one.

See *write_measurement()* for other parameters.

**write_measurements_odc_xarray**(*dataset*, *nodata=None*, *overviews=(8, 16, 32)*, *overview_resampling=Resampling.average*, *expand_valid_data=True*, *file_id=None*)

Write measurements from an ODC `xarray.Dataset`

The main requirement is that the Dataset contains a CRS attribute and X/Y or lat/long dimensions and coordinates. These are used to create an ODC GeoBox.

**Parameters** **dataset** (*xarray.Dataset*) – an xarray dataset (as returned by `datacube.Datacube.load()` and other methods)

See *write_measurement()* for other parameters.

**write_thumbnail**(*red*, *green*, *blue*, *resampling=Resampling.average*, *static_stretch=None*, *percentile_stretch=(2, 98)*, *scale_factor=10*, *kind=None*, *path=None*)

Write a thumbnail for the dataset using the given measurements (specified by name) as r/g/b.

(the measurements must already have been written.)

A linear stretch is performed on the colour. By default this is a dynamic 2% stretch (the 2% and 98% percentile values of the input). The static_stretch parameter will override this with a static range of values.

**Parameters**

- **red** (str) – Name of measurement to put in red band
- **green** (str) – Name of measurement to put in green band
- **blue** (str) – Name of measurement to put in blue band
- **kind** (Optional[str]) – If you have multiple thumbnails, you can specify the 'kind' name to distinguish them (it will be put in the filename). Eg. GA's ARD has two thumbnails, one of kind `nbar` and one of `nbart`.
- **scale_factor** (int) – How many multiples smaller to make the thumbnail.
- **percentile_stretch** (Tuple[int, int]) – Upper/lower percentiles to stretch by
- **resampling** (Resampling) – rasterio `rasterio.enums.Resampling` method to use.
- **static_stretch** (Optional[Tuple[int, int]]) – Use a static upper/lower value to stretch by instead of dynamic stretch.

**write_thumbnail_singleband**(*measurement*, *bit=None*, *lookup_table=None*, *kind=None*)

Write a singleband thumbnail out, taking in an input measurement and outputting a JPG with appropriate settings.

**Parameters**

- **measurement** (str) – Name of measurement
- **kind** (Optional[str]) – If you have multiple thumbnails, you can specify the 'kind' name to distinguish them (it will be put in the filename). Eg. GA's ARD has two thumbnails, one of kind `nbar` and one of `nbart`.

EITHER:

- Use a bit (int) as the value to scale from black to white to i.e., 0 will be BLACK and bit will be WHITE, with a linear scale between,:

```
p.write_thumbnail_singleband("blue", bit=1)
```

OR:

- Provide a lookup_table (dict) of int (key) [R, G, B] (value) fields to make the image with.:

```
p.write_thumbnail_singleband(
    "blue", lookup_table={1: (0, 0, 255)}
)
```

CHAPTER

# TWELVE

# READING/WRITING YAMLS

Methods for parsing and outputting EO3 docs as a *eodatasets3.DatasetDoc*

## 12.1 Parsing

eodatasets3.serialise.**from_path**(*path*, *skip_validation=False*)
Parse an EO3 document from a filesystem path

**Parameters**

- **path** (Path) – Filesystem path
- **skip_validation** – Optionally disable validation (it's faster, but I hope your doc is structured correctly)

**Return type** *DatasetDoc*

eodatasets3.serialise.**from_doc**(*doc*, *skip_validation=False*, *normalise_properties=False*)
Parse a dictionary into an EO3 dataset.

By default it will validate it against the schema, which will result in far more useful error messages if fields are missing.

**Parameters**

- **doc** (Dict) – A dictionary, such as is returned from yaml.load or json.load
- **skip_validation** – Optionally disable validation (it's faster, but I hope your doc is structured correctly)

**Return type** *DatasetDoc*

## 12.2 Writing

eodatasets3.serialise.**to_path**(*path*, **ds*)
Output dataset(s) as a formatted YAML to a local path

(multiple datasets will result in a multi-document yaml file)

eodatasets3.serialise.**to_stream**(*stream*, **ds*)
Output dataset(s) as a formatted YAML to an output stream

(multiple datasets will result in a multi-document yaml file)

eodatasets3.serialise.**to_doc**(*d*)

Serialise a DatasetDoc to a dict

If you plan to write this out as a yaml file on disk, you're better off with one of our formatted writers: *to_stream()*, *to_path()*.

**Return type** Dict

CHAPTER

# THIRTEEN

# NAME GENERATION API

You may want to use the name generation alone, for instance to tell if a dataset has already been written before you assemble it.

eodatasets3.**namer**(*properties=None*, ***, *collection_prefix=None*, *conventions='default'*)

Create a naming instance of the given conventions.

Conventions: 'default', 'dea', 'deafrica', ...

You usually give it existing properties, but you can use the return value's `.metadata` field to set properties afterwards.

**Return type** *NamingConventions*

**class** eodatasets3.**NamingConventions**(*properties*, *base_product_uri=None*, *required_fields=()*, *dataset_separator_field=None*, *allow_unknown_abbreviations=True*)

A generator of names for products, data labels, file paths, urls, etc.

These are generated based on a given set of naming conventions, but a user can manually override any properties to avoid generation.

Create an instance by calling *eodatasets3.namer()*:

```
from eodatasets3 import namer

properties = {
    'eo:platform': 'sentinel-2a',
    'eo:instrument': 'MSI',
    'odc:product_family': 'level1',
}
n = namer(properties, conventions='default')
print(n.product_name)
```

```
s2am_level1
```

**Note:** You may want to use an *eodatasets3.DatasetDoc* instance rather than a dict for properties, to get convenience methods such as `.platform = 'sentinel-2a'`, *.properties*, automatic property normalisation etc.

See *the naming section* for an example.

Fields are lazily generated when accessed using the underlying metadata properties, but you can manually set any field to avoid generation:

```
>>> from eodatasets3 import DatasetDoc
>>> p = DatasetDoc()
>>> p.platform = 'landsat-7'
>>> p.product_family = 'nbar'
>>>
>>> n = namer(conventions='default', properties=p)
>>> n.product_name
'ls7_nbar'
>>> # Manually override the abbreviation:
>>> n.platform_abbreviated = 'ls'
>>> n.product_name
'ls_nbar'
>>> # Or manually set the entire product name to avoid generation:
>>> n.product_name = 'custom_nbar_albers'
```

In order to calculate paths, give it a collection prefix. This can be a `Path object` for local files, or a URL str for remote.

```
>>> p.datetime = datetime(2014, 4, 5)
>>> collection = "s3://dea-public-data-dev/collections"
>>> n = namer(conventions='default', properties=p, collection_prefix=collection)
>>> n.dataset_location
's3://dea-public-data-dev/collections/ls7_nbar/2014/04/05/'
>>> n.metadata_file
'ls7_nbar_2014-04-05.odc-metadata.yaml'
```

All fields named `*_file` are filenames inside (relative to) the `self.dataset_location`.

```
>>> n.resolve_file('thumbnail.jpg')
's3://dea-public-data-dev/collections/ls7_nbar/2014/04/05/thumbnail.jpg'
```

**base_product_uri**
The default base URI used in product URI generation

Example: `https://collections.dea.ga.gov.au/`

**checksum_file: str**
The name of a checksum file

**property collection_path: Optional[pathlib.Path]**
Get the collection prefix as a Path on the local filesystem, if possible.

**Return type** Optional[Path]

**collection_prefix: Optional[str] = None**
The prefix where all files are stored, as a URI.

Eg. `'file:///my/dataset/collections'`

(used if dataset_location is generated)

**dataset_folder: str**
The full folder offset from the collection_prefix.

Example: `'ga_ls8c_ones_3/090/084/2016/01/21'`

(used if dataset_location is generated)

**dataset_location:** *str*
The full uri of the dataset as indexed into ODC.

**All inner document paths are relative to this.**

Eg. `s3://dea-public-data/ga_ls_fc_3/2-5-0/091/086/2020/04/04/ga_ls_fc_091086_2020-04-04.odc-metadata.yaml`

(Defaults to the metadata path inside the dataset_folder)

**property dataset_path:** *Optional[pathlib.Path]*
Get the dataset location as a Path on the local filesystem, if possible.

**Raises** *ValueError* – if the current dataset is not in a file:// location.

**Return type** *Optional*[*Path*]

**filename**(*file_id*, *suffix*)
Make a file name according to the current naming conventions' file pattern.

All filenames have a file_id (eg. "odc-metadata" or "") and a suffix (eg. "yaml")

Returned file paths are expected to be relative to the `self.dataset_location`

**Return type** *str*

**filename_pattern:** *str* = '{n.dataset_label}{file_id}.{suffix}'
The pattern for generating file names.

The pattern is in python's `str.format()` syntax, with fields `{file_id}` and `{suffix}`

The namer instance is readable from `{n}`.

**instrument_abbreviated:** *str*
Abbreviated form of the instrument, used in most other paths and names here.

For example, `ETM+` is usually abbreviated to `e`

**measurement_filename**(*measurement_name*, *suffix='tif'*, *file_id=None*)
Generate the path to a measurement for the current naming conventions.::

```
>> p.names.measurement_file('blue', 'tif')
'ga_ls8c_ones_3-0-0_090084_2016-01-21_final_blue.tif'
```

This is the filename inside the self.dataset_folder

**Return type** *str*

**metadata:** *eodatasets3.properties.Eo3Interface*
The underlying dataset properties used for generation.

**metadata_file:** *str*
The path or URL to the ODC metadata file.

(if relative, it's relative to self.dataset_location ... but could be absolute too)

Example: `'ga_ls8c_ones_3-0-0_090084_2016-01-21_final.odc-metadata.yaml'`

**platform_abbreviated:** *str*
Abbreviated form of the platform, used in most other paths and names here.

For example, `landsat-7` is usually abbreviated to `ls7`

**producer_abbreviated:** *str*
Abbreviated form of the producer of the dataset (the producing organisation)

For example, "ga.gov.au" is abbreviated to "ga"

**product_name:** *str*
Product name for ODC

**product_uri:** *str*
Identifier URL for the product (This is seen as a global id for the product, unlike the plain product name. It doesn't have to resolve to a real path)

Eg. `https://collections.dea.ga.gov.au/product/ga_ls8c_ard_3`

**region_folder**
The region portion of dataset_folder

By default, it will split the region code in half

Eg. `'012/094'`

**resolve_file**(*path*)
Convert the given file offset to a fully qualified URL within the dataset location.

**Return type** str

**resolve_path**(*path*)
Convert the given file offset (inside the dataset location) to a Path on the local filesystem (if possible).

**Raises** ValueError – if the current dataset is not in a file:// location.

**Return type** Path

**thumbnail_filename**(*kind=None*, *suffix='jpg'*)
Get a thumbnail file path (optionally with the given kind and/or suffix.)

**Return type** str

**time_folder**
The time portion of dataset_folder

By default, it will be `"%Y/%m/%d"` of the dataset's `'datetime'` property.

Eg. `'2019/03/12'`

CHAPTER

# FOURTEEN

# EO METADATA API

**class** eodatasets3.properties.**Eo3Interface**

These are convenience properties for common metadata fields. They are available on DatasetAssemblers and within other naming APIs.

(This is abstract. If you want one of these of your own, you probably want to create an *eodatasets3.DatasetDoc*)

**property collection_number: int**

The version of the collection.

Eg.:

```
metadata:
  product_family: wofs
  dataset_version: 1.6.0
  collection_number: 3
```

**Return type** int

**property constellation: str**

Constellation. Eg `sentinel-2`.

**Return type** str

**property dataset_version: str**

The version of the dataset.

Typically digits separated by a dot. Eg. *1.0.0*

The first digit is usually the collection number for this 'producer' organisation, such as USGS Collection 1 or GA Collection 3.

**Return type** str

**property datetime: datetime.datetime**

The searchable date and time of the assets. (Default to UTC if not specified)

**Return type** datetime

**datetime_**

alias of datetime.datetime

**property datetime_range: Tuple[datetime.datetime, datetime.datetime]**

An optional date range for the dataset.

The `datetime` is still mandatory when this is set.

This field is a shorthand for reading/setting the datetime-range stac 0.6 extension properties: `dtr:start_datetime` and `dtr:end_datetime`

**Return type** Tuple[datetime, datetime]

**property instrument: str**
Name of instrument or sensor used (e.g., MODIS, ASTER, OLI, Canon F-1).

Shorthand for `eo:instrument` property

**Return type** str

**property maturity: str**
The dataset maturity. The same data may be processed multiple times – becoming more mature – as new ancillary data becomes available.

Typical values (from least to most mature): `nrt` (near real time), `interim`, `final`

**Return type** str

**property platform: Optional[str]**
Unique name of the specific platform the instrument is attached to.

For satellites this would be the name of the satellite (e.g., `landsat-8`, `sentinel-2a`), whereas for drones this would be a unique name for the drone.

In derivative products, multiple platforms can be specified with a comma: `landsat-5,landsat-7`.

Shorthand for `eo:platform` property

**Return type** Optional[str]

**property platforms: Set[str]**
Get platform as a set (containing zero or more items).

In EO3, multiple platforms are specified by comma-separating them.

**Return type** Set[str]

**property processed: datetime.datetime**
When the dataset was created (Defaults to UTC if not specified)

Shorthand for the `odc:processing_datetime` field

**Return type** datetime

**processed_now()**
Shorthand for when the dataset was processed right now on the current system.

**property producer: str**
Organisation that produced the data.

eg. `usgs.gov` or `ga.gov.au`

Shorthand for `odc:producer` property

**Return type** str

**property product_family: str**
The identifier for this "family" of products, such as `ard`, `level1` or `fc`. It's used for grouping similar products together.

They products in a family are usually produced the same way but have small variations: they come from different sensors, or are written in different projections, etc.

`ard` family of products: `ls7_ard`, `ls5_ard` ....

On older versions of Open Data Cube this was called `product_type`.

Shorthand for `odc:product_family` property.

**Return type** str

**property product_maturity: str**
Classification: is this a 'provisional' or 'stable' release of the product?

**Return type** str

**property product_name: Optional[str]**
The ODC product name

**Return type** Optional[str]

**property region_code: Optional[str]**
The "region" of acquisition. This is a platform-agnostic representation of things like the Landsat Path+Row. Datasets with the same Region Code will *roughly* (but usually not *exactly*) cover the same spatial footprint.

It's generally treated as an opaque string to group datasets and process as stacks.

For Landsat products it's the concatenated `{path}{row}` (both numbers formatted to three digits).

For Sentinel 2, it's the MGRS grid (TODO presumably?).

Shorthand for `odc:region_code` property.

**Return type** Optional[str]

CHAPTER

# FIFTEEN

# MISC TYPES

**class** eodatasets3.**DatasetDoc**(*id=None*, *label=None*, *product=None*, *locations=None*, *crs=None*, *geometry=None*, *grids=None*, *properties=NOTHING*, *measurements=None*, *accessories=NOTHING*, *lineage=NOTHING*)

An EO3 dataset document

Includes *Eo3Interface* methods for metadata access:

```
>>> p = DatasetDoc()
>>> p.platform = 'LANDSAT_8'
>>> p.processed = '2018-04-03'
>>> p.properties['odc:processing_datetime']
datetime.datetime(2018, 4, 3, 0, 0, tzinfo=datetime.timezone.utc)
```

**id:** uuid.UUID
Dataset UUID

**label:** str
Human-readable identifier for the dataset

**product:** eodatasets3.model.ProductDoc
The product name (local) and/or url (global)

**locations:** List[str]
Location(s) where this dataset is stored.

(ODC supports multiple locations when the same dataset is stored in multiple places)

They are fully qualified URIs (`file://...`, ``https://..., s3://...`)

All other paths in the document (measurements, accessories) are relative to the chosen location.

**crs:** str
CRS string. Eg. `epsg:3577`

**geometry:** shapely.geometry.base.BaseGeometry
Shapely geometry of the valid data coverage

(it must contain all non-empty pixels of the image)

**grids:** Dict[str, eodatasets3.model.GridDoc]
Grid specifications for measurements

**properties:** *eodatasets3.properties.Eo3Dict*
Raw properties

**measurements:** Dict[str, eodatasets3.model.MeasurementDoc]
Loadable measurements of the dataset

**accessories:** Dict[str, eodatasets3.model.AccessoryDoc]
References to accessory files

Such as thumbnails, checksums, other kinds of metadata files.

(any files included in the dataset that are not measurements)

**lineage:** Dict[str, List[uuid.UUID]]
Links to source dataset uuids

class eodatasets3.**Eo3Dict**(*properties=None*, *normalise_input=True*)
This acts like a dictionary, but will normalise known properties (consistent case, types etc) and warn about common mistakes.

It wraps an inner dictionary. By default it will normalise the fields in the input dictionary on creation, but you can disable this with *normalise_input=False*.

**normalise_and_set**(*key*, *value*, *allow_override=True*, *expect_override=False*)
Set a property with the usual normalisation.

This has some options that are not available on normal dictionary item setting (`self[key] = val`)

The default behaviour of this class is very conservative in order to catch common errors of users. You can loosen the settings here.

**Parameters**

- **allow_override** – Is it okay to overwrite an existing value? (if not, error will be thrown)
- **expect_override** – We expect to overwrite a property, so don't produce a warning or error.

class eodatasets3.**GridSpec**(*shape*, *transform*, *crs=None*)
The grid spec defines the coordinates/transform and size of pixels of a measurment.

The easiest way to create one is use the `GridSpec.from_*()` class methods, such as `GridSpec.from_path(my_image_path)`.

To create one manually:

```
>>> from eodatasets3 import GridSpec
>>> from affine import Affine
>>> from rasterio.crs import CRS
>>> g = GridSpec(shape=(7721, 7621),
...              transform=Affine(30.0, 0.0, 241485.0, 0.0, -30.0, -2281485.0),
...              crs=CRS.from_epsg(32656))
>>> # Numbers copied from equivalent rio dataset.bounds call.
>>> g.bounds
BoundingBox(left=241485.0, bottom=-2513115.0, right=470115.0, top=-2281485.0)
>>> g.resolution_yx
(30.0, 30.0)
```

**shape:** Tuple[int, int]

**transform:** affine.Affine

**crs:** rasterio.crs.CRS

classmethod **from_dataset_doc**(*ds*, *grid='default'*)
Create from an existing parsed metadata document

**Parameters** **grid** – Grid name to read, if not the default.

**Return type** *GridSpec*

**classmethod from_rio**(*dataset*)
Create from an open rasterio dataset

**Return type** *GridSpec*

**classmethod from_odc_xarray**(*dataset*)
Create from an ODC xarray

**Return type** *GridSpec*

**classmethod from_path**(*path*)
Create from the spec of a (rio-readable) filesystem path or url

**Return type** *GridSpec*

**property bounds**
Get bounding box.

**class** eodatasets3.**IfExists**(*value*)
Enum: what to do when output already exists?

**Skip = 1**
Skip the dataset

**Overwrite = 2**
Overwrite the existing dataset

**ThrowError = 3**
Throw an error

**exception** eodatasets3.**IncompleteDatasetError**(*validation*)
Raised when a dataset is missing essential things and so cannot be written.

(such as mandatory metadata)

**class** eodatasets3.**ValidDataMethod**(*value*)
How to calculate the valid data geometry for an image?

**thorough = 1**
Vectorize the full valid pixel mask as-is.

In some circumstances this can be very slow. *filled* may be safer.

**filled = 2**
Fill holes in the valid pixel mask before vectorizing.

(Potentially much faster than `thorough` if there's many small nodata holes, as they will create many tiny polygons. *slightly* slower if no holes exist.)

**convex_hull = 3**
Take convex-hull of valid pixel mask before vectorizing.

This is much slower than `filled`, but will work in cases where you have a lot of internal geometry that aren't holes. Such as SLC-Off Landsat 7 data.

Requires 'scikit-image' dependency.

**bounds = 4**
Use the image file bounds, ignoring actual pixel values.

# PYTHON MODULE INDEX

e

# INDEX

## Symbols



## A



## B



## C



## D



## E



## F



## G

## I



## L



## M



## N



## O



## P



## R



## S



## T

## V



## W