eodatasets3

Geoscience Australia

May 02, 2023

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

CHAPTER

ELEVEN

DATASET ASSEMBLER CLASS REFERENCE

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*)

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.

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
```

property constellation: str

Constellation. Eg `sentinel-2`.

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.

property datetime: datetime

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

datetime_

alias of `datetime`

property datetime_range: Tuple[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`

property instrument: str

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

Shorthand for `eo:instrument` property

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`

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

property platforms: Set[str]

Get platform as a set (containing zero or more items).

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

property processed: datetime

When the dataset was created (Defaults to UTC if not specified)

Shorthand for the `odc:processing_datetime` field

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

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.

property product_maturity: str

Classification: is this a 'provisional' or 'stable' release of the product?

property product_name: Optional[str]

The ODC product name

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.

CHAPTER

FIFTEEN

MISC TYPES

INDEX