
rasterio Documentation

Sean Gillies

Oct 20, 2023

CONTENTS

1	Introduction	3
1.1	Philosophy	3
1.2	Rasterio license	3
2	Installation	5
2.1	Easy installation	5
2.2	Advanced installation	5
3	Python Quickstart	7
3.1	Opening a dataset in reading mode	7
3.2	Dataset attributes	9
3.3	Dataset georeferencing	9
3.4	Reading raster data	10
3.5	Spatial indexing	11
3.6	Creating data	11
3.7	Opening a dataset in writing mode	12
3.8	Saving raster data	13
4	Command Line User Guide	15
4.1	creation options	16
4.2	blocks	16
4.3	bounds	17
4.4	calc	18
4.5	clip	18
4.6	convert	18
4.7	edit-info	19
4.8	info	20
4.9	insp	21
4.10	mask	22
4.11	merge	22
4.12	overview	22
4.13	rasterize	23
4.14	rm	23
4.15	sample	23
4.16	shapes	24
4.17	stack	24
4.18	transform	24
4.19	warp	25
4.20	Rio Plugins	25
4.21	Other commands?	26

5	Advanced Topics	27
5.1	Using rio-calc	27
5.2	Color	29
5.3	Concurrent processing	31
5.4	GDAL Option Configuration	34
5.5	Advanced Datasets	35
5.6	Error Handling	36
5.7	Vector Features	37
5.8	Filling nodata areas	39
5.9	Georeferencing	39
5.10	Options	41
5.11	Interoperability	42
5.12	Masking a raster using a shapefile	43
5.13	Nodata Masks	44
5.14	In-Memory Files	51
5.15	Migrating to Rasterio 1.0	53
5.16	Overviews	57
5.17	Plotting	58
5.18	Profiles and Writing Files	64
5.19	Reading Datasets	65
5.20	Reprojection	68
5.21	Resampling	72
5.22	Switching from GDAL's Python bindings	74
5.23	Tagging datasets and bands	80
5.24	Transforms	81
5.25	Virtual Warping	83
5.26	Virtual Filesystems	86
5.27	Windowed reading and writing	87
5.28	Writing Datasets	92
6	Python API Reference	95
6.1	rasterio package	95
7	Contributing	115
7.1	Code of Conduct	115
7.2	Rights	115
7.3	Issue Conventions	115
7.4	Design Principles	116
7.5	Dataset Objects	116
7.6	Path Objects	116
7.7	Band Objects	116
7.8	GDAL Context	116
7.9	Git Conventions	117
7.10	Conventions	117
7.11	New Containerized Development Environment	117
7.12	Historical Development Environment	117
7.13	Additional Information	119
8	Frequently Asked Questions	121
8.1	What does “RasterioIOError: file.ecw not recognized as a supported file format.” mean?	121
8.2	Where is “ERROR 4: Unable to open EPSG support file gcs.csv” coming from and what does it mean?	121
8.3	Why can't rasterio find proj.db (rasterio versions < 1.2.0)?	122
8.4	Why can't rasterio find proj.db (rasterio from PyPI versions >= 1.2.0)?	122
9	Indices and Tables	123

Python Module Index	125
Index	127

Geographic information systems use GeoTIFF and other formats to organize and store gridded raster datasets such as satellite imagery and terrain models. Rasterio reads and writes these formats and provides a Python API based on Numpy N-dimensional arrays and GeoJSON.

Here's an example program that extracts the GeoJSON shapes of a raster's valid data footprint.

```
import rasterio
import rasterio.features
import rasterio.warp

with rasterio.open('example.tif') as dataset:

    # Read the dataset's valid data mask as a ndarray.
    mask = dataset.dataset_mask()

    # Extract feature shapes and values from the array.
    for geom, val in rasterio.features.shapes(
        mask, transform=dataset.transform):

        # Transform shapes from the dataset's own coordinate
        # reference system to CRS84 (EPSG:4326).
        geom = rasterio.warp.transform_geom(
            dataset.crs, 'EPSG:4326', geom, precision=6)

    # Print GeoJSON shapes to stdout.
    print(geom)
```

The output of the program:

```
{'type': 'Polygon', 'coordinates': [[(-77.730817, 25.282335), ...]]}
```

Rasterio supports Python versions 3.6 or higher.

INTRODUCTION

1.1 Philosophy

Before Rasterio there was one Python option for accessing the many different kind of raster data files used in the GIS field: the Python bindings distributed with the [Geospatial Data Abstraction Library](#), [GDAL](#). These bindings extend Python, but provide little abstraction for GDAL's C API. This means that Python programs using them tend to read and run like C programs. For example, GDAL's Python bindings require users to watch out for dangling C pointers, potential crashers of programs. This is bad: among other considerations we've chosen Python instead of C to avoid problems with pointers.

What would it be like to have a geospatial data abstraction in the Python standard library? One that used modern Python language features and idioms? One that freed users from concern about dangling pointers and other C programming pitfalls? Rasterio's goal is to be this kind of raster data library – expressing GDAL's data model using fewer non-idiomatic extension classes and more idiomatic Python types and protocols, while performing as fast as GDAL's Python bindings.

High performance, lower cognitive load, cleaner and more transparent code. This is what Rasterio is about.

1.2 Rasterio license

Copyright (c) 2016, MapBox All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Mapbox nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

INSTALLATION

Installation of the Rasterio package is complicated by its dependency on libgdal and other C libraries. There are easy installations paths and an advanced installation path.

2.1 Easy installation

Rasterio has several [extension modules](#) which link against libgdal. This complicates installation. Binary distributions (wheels) containing libgdal and its own dependencies are available from the Python Package Index and can be installed using pip.

```
pip install rasterio
```

These wheels are mainly intended to make installation easy for simple applications, not so much for production. They are not tested for compatibility with all other binary wheels, conda packages, or QGIS, and omit many of GDAL's optional format drivers.

Many users find Anaconda and conda-forge a good way to install Rasterio and get access to more optional format drivers (like TileDB and others).

Rasterio 1.4 requires Python 3.9 or higher and GDAL 3.3 or higher.

2.2 Advanced installation

Once GDAL and its dependencies are installed on your computer (how to do this is documented at <https://gdal.org>) Rasterio can be built and installed using setuptools or pip. If your GDAL installation provides the `gdal-config` program, the process is simpler.

Without pip:

```
GDAL_CONFIG=/path/to/gdal-config python setup.py install
```

With pip (version `>= 22.3` is required):

```
python -m pip install --user -U pip
GDAL_CONFIG=/path/to/gdal-config python -m pip install --user --no-binary rasterio,
↳ rasterio
```

These are pretty much equivalent. Pip will use setuptools as the build backend. If the `gdal-config` program is on your executable path, then you don't need to set the environment variable.

Without `gdal-config` you will need to configure header and library locations for the build in another way. One way to do this is to create a `setup.cfg` file in the source directory with content like this:

```
[build_ext]
include_dirs = C:/vcpkg/installed/x64-windows/include
libraries = gdal
library_dirs = C:/vcpkg/installed/x64-windows/lib
```

This is the approach taken by Rasterio's [wheel-building workflow](#). With this file in place you can run either `python setup.py install` or `python -m pip install --user ..`

You can also pass those three values on the command line following the [setuptools documentation](#). However, the `setup.cfg` approach is easier.

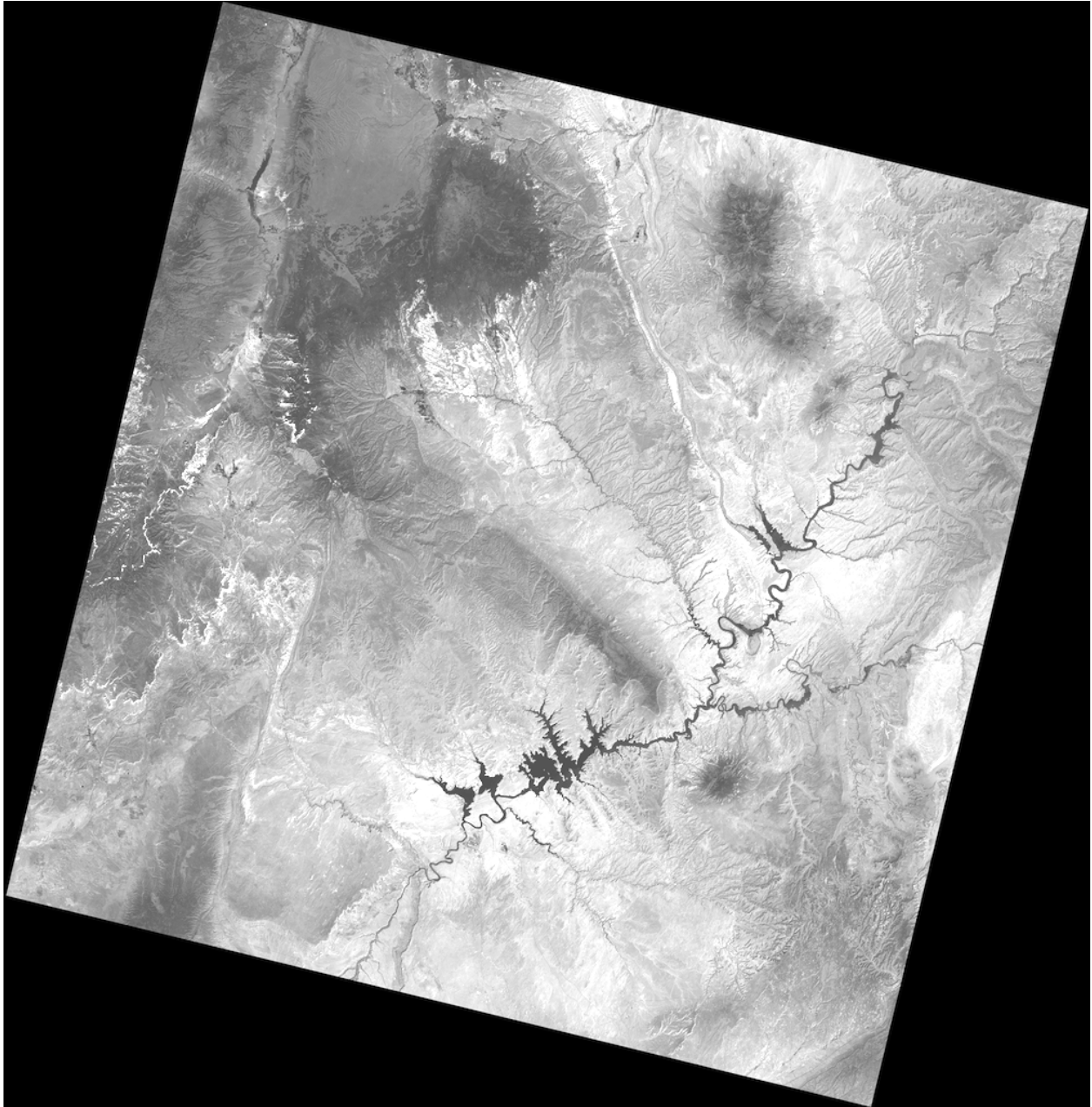
PYTHON QUICKSTART

Reading and writing data files is a spatial data programmer's bread and butter. This document explains how to use Rasterio to read existing files and to create new files. Some advanced topics are glossed over to be covered in more detail elsewhere in Rasterio's documentation. Only the GeoTIFF format is used here, but the examples do apply to other raster data formats. It is presumed that Rasterio has been *installed*.

3.1 Opening a dataset in reading mode

Consider a GeoTIFF file named `example.tif` with 16-bit Landsat 8 imagery covering a part of the United States's Colorado Plateau¹. Because the imagery is large (70 MB) and has a wide dynamic range it is difficult to display it in a browser. A rescaled and dynamically squashed version is shown below.

¹ "example.tif" is an alias for band 4 of Landsat scene LC80370342016194LGN00.



Import rasterio to begin.

```
>>> import rasterio
```

Next, open the file.

```
>>> dataset = rasterio.open('example.tif')
```

Rasterio's `open()` function takes a path string or path-like object and returns an opened dataset object. The path may point to a file of any supported raster format. Rasterio will open it using the proper GDAL format driver. Dataset objects have some of the same attributes as Python file objects.

```
>>> dataset.name
```

(continues on next page)

(continued from previous page)

```
'example.tif'
>>> dataset.mode
'r'
>>> dataset.closed
False
```

3.2 Dataset attributes

Properties of the raster data stored in the example GeoTIFF can be accessed through attributes of the opened dataset object. Dataset objects have bands and this example has a band count of 1.

```
>>> dataset.count
1
```

A dataset band is an array of values representing the partial distribution of a single variable in 2-dimensional (2D) space. All band arrays of a dataset have the same number of rows and columns. The variable represented by the example dataset's sole band is Level-1 digital numbers (DN) for the Landsat 8 Operational Land Imager (OLI) band 4 (wavelengths between 640-670 nanometers). These values can be scaled to radiance or reflectance values. The array of DN values is 7731 columns wide and 7871 rows high.

```
>>> dataset.width
7731
>>> dataset.height
7871
```

Some dataset attributes expose the properties of all dataset bands via a tuple of values, one per band. To get a mapping of band indexes to variable data types, apply a dictionary comprehension to the `zip()` product of a dataset's `DatasetReader.indexes` and `DatasetReader.dtypes` attributes.

```
>>> {i: dtype for i, dtype in zip(dataset.indexes, dataset.dtypes)}
{1: 'uint16'}
```

The example file's sole band contains unsigned 16-bit integer values. The GeoTIFF format also supports signed integers and floats of different size.

3.3 Dataset georeferencing

A GIS raster dataset is different from an ordinary image; its elements (or “pixels”) are mapped to regions on the earth's surface. Every pixels of a dataset is contained within a spatial bounding box.

```
>>> dataset.bounds
BoundingBox(left=358485.0, bottom=4028985.0, right=590415.0, top=4265115.0)
```

Our example covers the world from 358485 meters (in this case) to 590415 meters, left to right, and 4028985 meters to 4265115 meters bottom to top. It covers a region 231.93 kilometers wide by 236.13 kilometers high.

The value of `DatasetReader.bounds` attribute is derived from a more fundamental attribute: the dataset's geospatial transform.

```
>>> dataset.transform
Affine(30.0, 0.0, 358485.0,
       0.0, -30.0, 4265115.0)
```

A dataset's `DatasetReader.transform` is an affine transformation matrix that maps pixel locations in (col, row) coordinates to (x, y) spatial positions. The product of this matrix and `(0, 0)`, the column and row coordinates of the upper left corner of the dataset, is the spatial position of the upper left corner.

```
>>> dataset.transform * (0, 0)
(358485.0, 4265115.0)
```

The position of the lower right corner is obtained similarly.

```
>>> dataset.transform * (dataset.width, dataset.height)
(590415.0, 4028985.0)
```

But what do these numbers mean? 4028985 meters from where? These coordinate values are relative to the origin of the dataset's coordinate reference system (CRS).

```
>>> dataset.crs
CRS.from_epsg(32612)
```

[EPSG:32612](#) identifies a particular coordinate reference system: [UTM](#) zone 12N. This system is used for mapping areas in the Northern Hemisphere between 108 and 114 degrees west. The upper left corner of the example dataset, `(358485.0, 4265115.0)`, is 141.5 kilometers west of zone 12's central meridian (111 degrees west) and 4265 kilometers north of the equator.

Between the `DatasetReader.crs` and `DatasetReader.transform` attributes, the georeferencing of a raster dataset is described and the dataset can be compared to other GIS datasets.

3.4 Reading raster data

Data from a raster band can be accessed by the band's index number. Following the GDAL convention, bands are indexed from 1.

```
>>> dataset.indexes
(1,)
>>> band1 = dataset.read(1)
```

The `DatasetReader.read()` method returns a [numpy.ndarray](#).

```
>>> band1
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint16)
```

Values from the array can be addressed by their row, column index.


```
>>> band1[dataset.height // 2, dataset.width // 2]
17491
```

3.5 Spatial indexing

Datasets have an `DatasetReader.index()` method for getting the array indices corresponding to points in georeferenced space. To get the value for the pixel 100 kilometers east and 50 kilometers south of the dataset's upper left corner, do the following.

```
>>> x, y = (dataset.bounds.left + 100000, dataset.bounds.top - 50000)
>>> row, col = dataset.index(x, y)
>>> row, col
(1666, 3333)
>>> band1[row, col]
7566
```

To get the spatial coordinates of a pixel, use the dataset's `DatasetReader.xy()` method. The coordinates of the center of the image can be computed like this.

```
>>> dataset.xy(dataset.height // 2, dataset.width // 2)
(476550.0, 4149150.0)
```

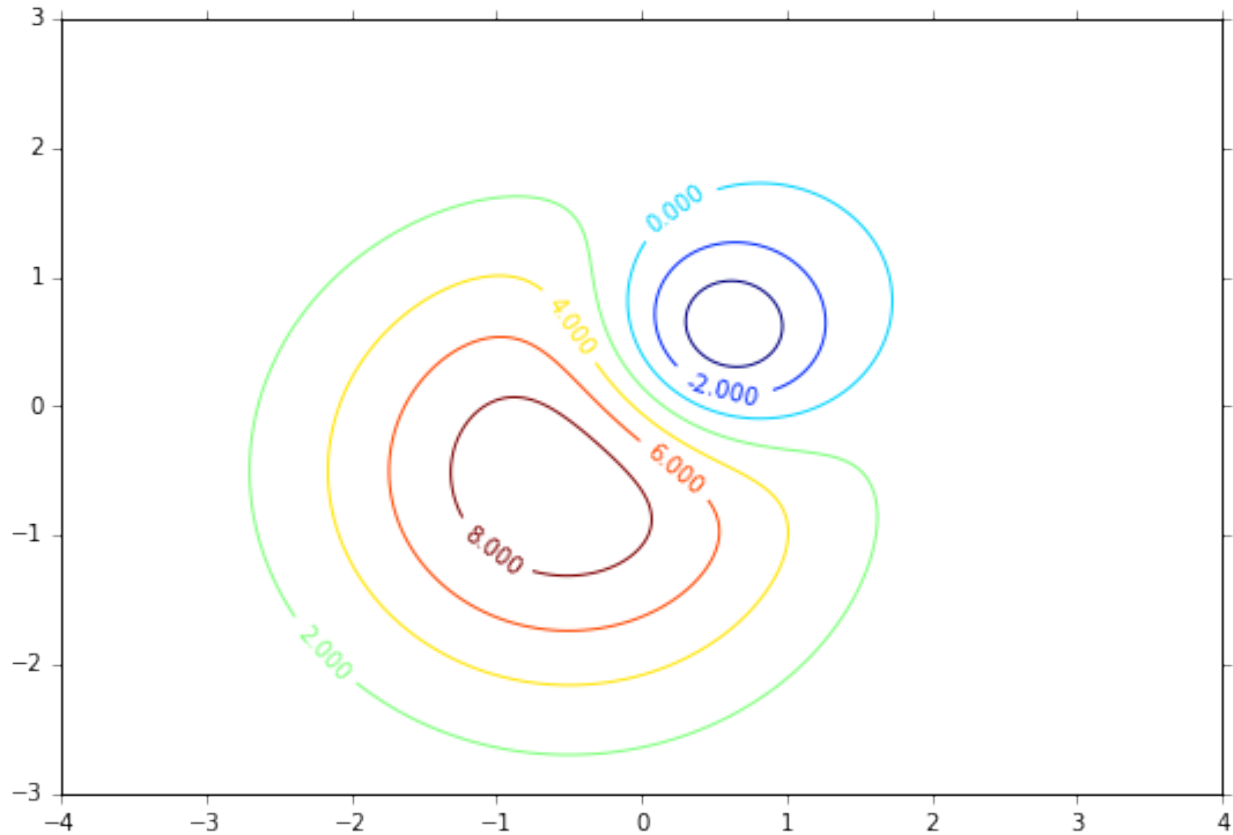
3.6 Creating data

Reading data is only half the story. Using Rasterio dataset objects, arrays of values can be written to a raster data file and thus shared with other GIS applications such as QGIS.

As an example, consider an array of floating point values representing, e.g., a temperature or pressure anomaly field measured or modeled on a regular grid, 240 columns by 180 rows. The first and last grid points on the horizontal axis are located at 4.0 degrees west and 4.0 degrees east longitude, the first and last grid points on the vertical axis are located at 3 degrees south and 3 degrees north latitude.

```
>>> import numpy as np
>>> x = np.linspace(-4.0, 4.0, 240)
>>> y = np.linspace(-3.0, 3.0, 180)
>>> X, Y = np.meshgrid(x, y)
>>> Z1 = np.exp(-2 * np.log(2) * ((X - 0.5) ** 2 + (Y - 0.5) ** 2) / 1 ** 2)
>>> Z2 = np.exp(-3 * np.log(2) * ((X + 0.5) ** 2 + (Y + 0.5) ** 2) / 2.5 ** 2)
>>> Z = 10.0 * (Z2 - Z1)
```

The fictional field for this example consists of the difference of two Gaussian distributions and is represented by the array `Z`. Its contours are shown below.



3.7 Opening a dataset in writing mode

To save this array along with georeferencing information to a new raster data file, call `rasterio.open()` with a path to the new file to be created, 'w' to specify writing mode, and several keyword arguments.

- *driver*: the name of the desired format driver
- *width*: the number of columns of the dataset
- *height*: the number of rows of the dataset
- *count*: a count of the dataset bands
- *dtype*: the data type of the dataset
- *crs*: a coordinate reference system identifier or description
- *transform*: an affine transformation matrix, and
- *nodata*: a “nodata” value

The first 5 of these keyword arguments parametrize fixed, format-specific properties of the data file and are required when opening a file to write. The last 3 are optional.

In this example the coordinate reference system will be `'+proj=latlong'`, which describes an equirectangular coordinate reference system with units of decimal degrees. The proper affine transformation matrix can be computed from the matrix product of a translation and a scaling.

```
>>> from rasterio.transform import Affine
>>> res = (x[-1] - x[0]) / 240.0
>>> transform = Affine.translation(x[0] - res / 2, y[0] - res / 2) * Affine.scale(res,
↳ res)
>>> transform
Affine(0.03333333333333333, 0.0, -4.0166666666666666,
      0.0, 0.03333333333333333, -3.0166666666666666)
```

The upper left point in the example grid is at 3 degrees west and 2 degrees north. The raster pixel centered on this grid point extends $\text{res} / 2$, or 1/60 degrees, in each direction, hence the shift in the expression above.

A dataset for storing the example grid is opened like so

```
>>> new_dataset = rasterio.open(
...     '/tmp/new.tif',
...     'w',
...     driver='GTiff',
...     height=Z.shape[0],
...     width=Z.shape[1],
...     count=1,
...     dtype=Z.dtype,
...     crs='+proj=latlong',
...     transform=transform,
... )
```

Values for the *height*, *width*, and *dtype* keyword arguments are taken directly from attributes of the 2-D array, *Z*. Not all raster formats can support the 64-bit float values in *Z*, but the GeoTIFF format can.

3.8 Saving raster data

To copy the grid to the opened dataset, call the new dataset's `DatasetWriter.write()` method with the grid and target band number as arguments.

```
>>> new_dataset.write(Z, 1)
```

Then call the `DatasetWriter.close()` method to sync data to disk and finish.

```
>>> new_dataset.close()
```

Because Rasterio's dataset objects mimic Python's file objects and implement Python's context manager protocol, it is possible to do the following instead.

```
with rasterio.open(
    '/tmp/new.tif',
    'w',
    driver='GTiff',
    height=Z.shape[0],
    width=Z.shape[1],
    count=1,
    dtype=Z.dtype,
    crs='+proj=latlong',
    transform=transform,
```

(continues on next page)

(continued from previous page)

```
) as dst:  
    dst.write(Z, 1)
```

These are the basics of reading and writing raster data files. More features and examples are contained in the [advanced topics](#) section.

COMMAND LINE USER GUIDE

Rasterio's command line interface (CLI) is a program named “rio”¹.

The CLI allows you to build workflows using shell commands, either interactively at the command prompt or with a script. Many common cases are covered by CLI sub-commands and it is often more convenient to use a ready-made command as opposed to implementing similar functionality as a python script.

The rio program is developed using the [Click](#) framework. Its plugin system allows external modules to share a common namespace and handling of context variables.

```
$ rio --help
Usage: rio [OPTIONS] COMMAND [ARGS]...

  Rasterio command line interface.

Options:
  -v, --verbose          Increase verbosity.
  -q, --quiet            Decrease verbosity.
  --aws-profile TEXT     Select a profile from the AWS credentials file
  --aws-no-sign-requests Make requests anonymously
  --aws-requester-pays   Requester pays data transfer costs
  --version              Show the version and exit.
  --gdal-version         Show the version and exit.
  --help                Show this message and exit.

Commands:
  blocks    Write dataset blocks as GeoJSON features.
  bounds    Write bounding boxes to stdout as GeoJSON.
  calc      Raster data calculator.
  clip      Clip a raster to given bounds.
  convert    Copy and convert raster dataset.
  edit-info Edit dataset metadata.
  env       Print information about the Rasterio environment.
  gcps      Print ground control points as GeoJSON.
  info      Print information about a data file.
  insp      Open a data file and start an interpreter.
  mask      Mask in raster using features.
  merge     Merge a stack of raster datasets.
  overview  Construct overviews in an existing dataset.
```

(continues on next page)

¹ In some Linux distributions “rio” may instead refer to the command line Diamond Rio MP3 player controller. This conflict can be avoided by installing Rasterio in an isolated Python environment.

(continued from previous page)

```
rasterize  Rasterize features.
rm         Delete a dataset.
sample     Sample a dataset.
shapes     Write shapes extracted from bands or masks.
stack      Stack a number of bands into a multiband dataset.
transform  Transform coordinates.
warp       Warp a raster dataset.
```

Commands are shown below. See `--help` of individual commands for more details.

4.1 creation options

For commands that create new datasets, format specific creation options may also be passed using `--co`. For example, to tile a new GeoTIFF output file, add the following.

```
--co tiled=true --co blockxsize=256 --co blockysize=256
```

To compress it using the LZW method, add

```
--co compress=LZW
```

4.2 blocks

This command prints features describing a raster's internal blocks, which are used directly for raster I/O. These features can be used to visualize how a windowed operation would operate using those blocks.

Output features have two JSON encoded properties: `block` and `window`. `Block` is a two element array like `[0, 0]` describing the window's position in the input band's window layout. `Window` is a JSON serialization of rasterio's `Window` class like `{"col_off": 0, "height": 3, "row_off": 705, "width": 791}`.

Block windows are extracted from the dataset (all bands must have matching block windows) by default, or from the band specified using the `--bidx` option:

```
rio blocks --bidx 3 tests/data/RGB.byte.tif
```

By default a GeoJSON FeatureCollection is written. With the `--sequence` option a GeoJSON feature stream is written instead.

```
rio blocks tests/data/RGB.byte.tif --sequence
```

Output features are reprojected to OGC:CRS84 (WGS 84) unless the `--projected` flag is provided, which causes the output to be kept in the input datasource's coordinate reference system.

For more information on exactly what blocks and windows represent, see `rasterio._base.DatasetBase.block_windows()`.

4.3 bounds

The `bounds` command writes the bounding boxes of raster datasets to GeoJSON for use with, e.g., [geojsonio-cli](#).

```
$ rio bounds tests/data/RGB.byte.tif --indent 2
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          [
            [
              -78.898133,
              23.564991
            ],
            [
              -76.599438,
              23.564991
            ],
            [
              -76.599438,
              25.550874
            ],
            [
              -78.898133,
              25.550874
            ],
            [
              -78.898133,
              23.564991
            ]
          ]
        ],
        "type": "Polygon"
      },
      "properties": {
        "id": "0",
        "title": "tests/data/RGB.byte.tif"
      },
      "type": "Feature"
    }
  ],
  "type": "FeatureCollection"
}
```

Shoot the GeoJSON into a Leaflet map using `geojsonio-cli` by typing `rio bounds tests/data/RGB.byte.tif | geojsonio`.

4.4 calc

The `calc` command reads files as arrays, evaluates lisp-like expressions in their context, and writes the result as a new file. Members of the `numpy` module and arithmetic and logical operators are available builtin functions and operators. It is intended for simple calculations; any calculations requiring multiple steps is better done in Python using the Rasterio and Numpy APIs.

Input files may have different numbers of bands but should have the same number of rows and columns. The output file will have the same number of rows and columns as the inputs and one band per element of the expression result. An expression involving arithmetic operations on N-D arrays will produce a N-D array and result in an N-band output file.

The following produces a 3-band GeoTIFF with all values scaled by 0.95 and incremented by 2. In the expression, `(read 1)` evaluates to the first input dataset (3 bands) as a 3-D array.

```
$ rio calc "(+ 2 (* 0.95 (read 1)))" tests/data/RGB.byte.tif /tmp/out.tif
```

The following produces a 3-band GeoTIFF in which the first band is copied from the first band of the input and the next two bands are scaled (down) by the ratio of the first band's mean to their own means. The `--name` option is used to bind datasets to a name within the expression. `(take a 1)` gets the first band of the dataset named `a` as a 2-D array and `(asarray ...)` collects a sequence of 2-D arrays into a 3-D array for output.

```
$ rio calc "(asarray (take a 1) (* (take a 2) (/ (mean (take a 1)) (mean (take a 2)))) \
→(* (take a 3) (/ (mean (take a 1)) (mean (take a 3)))))" \
> --name a=tests/data/RGB.byte.tif /tmp/out.rgb.tif
```

The command above is also an example of a calculation that is far beyond the design of the `calc` command and something that could be done much more efficiently in Python.

4.5 clip

The `clip` command clips a raster using bounds input directly or from a template raster.

```
$ rio clip input.tif output.tif --bounds xmin ymin xmax ymax
$ rio clip input.tif output.tif --like template.tif
```

If using `--bounds`, values must be in coordinate reference system of input. If using `--like`, bounds will automatically be transformed to match the coordinate reference system of the input.

It can also be combined to read bounds of a feature dataset using Fiona:

```
$ rio clip input.tif output.tif --bounds $(fio info features.shp --bounds)
```

4.6 convert

The `convert` command copies and converts raster datasets to other data types and formats (similar to `gdal_translate`).

Data values may be linearly scaled when copying by using the `--scale-ratio` and `--scale-offset` options. Destination raster values are calculated as


```
dst = scale_ratio * src + scale_offset
```

For example, to scale uint16 data with an actual range of 0-4095 to 0-255 as uint8:

```
$ rio convert in16.tif out8.tif --dtype uint8 --scale-ratio 0.0625
```

You can use `-rgb` as shorthand for `-co photometric=rgb`.

4.7 edit-info

The `edit-info` command allows you edit a raster dataset's metadata, namely

- coordinate reference system
- affine transformation matrix
- nodata value
- tags
- color interpretation

A TIFF created by spatially-unaware image processing software like Photoshop or Imagemagick can be turned into a GeoTIFF by editing these metadata items.

For example, you can set or change a dataset's coordinate reference system to Web Mercator (EPSG:3857),

```
$ rio edit-info --crs EPSG:3857 example.tif
```

set its *affine transformation matrix*,

```
$ rio edit-info --transform "[300.0, 0.0, 101985.0, 0.0, -300.0, 2826915.0]" example.tif
```

or set its nodata value to, e.g., 0:

```
$ rio edit-info --nodata 0 example.tif
```

or set its color interpretation to red, green, blue, and alpha:

```
$ rio edit-info --colorinterp 1=red,2=green,3=blue,4=alpha example.tif
```

which can also be expressed as:

```
$ rio edit-info --colorinterp RGBA example.tif
```

See `rasterio.enums.ColorInterp` for a full list of supported color interpretations and the color docs for more information.

4.8 info

The `info` command prints structured information about a dataset.

```
$ rio info tests/data/RGB.byte.tif --indent 2
{
  "count": 3,
  "crs": "EPSG:32618",
  "dtype": "uint8",
  "driver": "GTiff",
  "bounds": [
    101985.0,
    2611485.0,
    339315.0,
    2826915.0
  ],
  "lnglat": [
    -77.75790625255473,
    24.561583285327067
  ],
  "height": 718,
  "width": 791,
  "shape": [
    718,
    791
  ],
  "res": [
    300.0379266750948,
    300.041782729805
  ],
  "nodata": 0.0
}
```

More information, such as band statistics, can be had using the `--verbose` option.

```
$ rio info tests/data/RGB.byte.tif --indent 2 --verbose
{
  "count": 3,
  "crs": "EPSG:32618",
  "stats": [
    {
      "max": 255.0,
      "mean": 44.434478650699106,
      "min": 1.0
    },
    {
      "max": 255.0,
      "mean": 66.02203484105824,
      "min": 1.0
    },
    {
      "max": 255.0,
      "mean": 71.39316199120559,
```

(continues on next page)

(continued from previous page)

```
    "min": 1.0
  }
],
"dtype": "uint8",
"driver": "GTiff",
"bounds": [
  101985.0,
  2611485.0,
  339315.0,
  2826915.0
],
"lnglat": [
  -77.75790625255473,
  24.561583285327067
],
"height": 718,
"width": 791,
"shape": [
  718,
  791
],
"res": [
  300.0379266750948,
  300.041782729805
],
"nodata": 0.0
}
```

4.9 insp

The `insp` command opens a dataset and an interpreter.

```
$ rio insp --ipython tests/data/RGB.byte.tif
Rasterio 0.32.0 Interactive Inspector (Python 2.7.10)
Type "src.meta", "src.read(1)", or "help(src)" for more information.
In [1]: print(src.name)
/path/rasterio/tests/data/RGB.byte.tif

In [2]: print(src.bounds)
BoundingBox(left=101985.0, bottom=2611485.0, right=339315.0, top=2826915.0)
```

4.10 mask

The `mask` command masks in pixels from all bands of a raster using features (masking out all areas not covered by features) and optionally crops the output raster to the extent of the features. Features are assumed to be in the same coordinate reference system as the input raster.

A common use case is masking in raster data by political or other boundaries.

```
$ rio mask input.tif output.tif --geojson-mask input.geojson
```

GeoJSON features may be provided using stdin or specified directly as first argument, and output can be cropped to the extent of the features.

```
$ rio mask input.tif output.tif --crop --geojson-mask - < input.geojson
```

The feature mask can be inverted to mask out pixels covered by features and keep pixels not covered by features.

```
$ rio mask input.tif output.tif --invert --geojson-mask input.geojson
```

4.11 merge

The `merge` command can be used to flatten a stack of identically structured datasets.

```
$ rio merge rasterio/tests/data/R*.tif merged.tif
```

4.12 overview

The `overview` command creates overviews stored in the dataset, which can improve performance in some applications.

The decimation levels at which to build overviews can be specified as a comma separated list

```
$ rio overview --build 2,4,8,16
```

or a base and range of exponents.

```
$ rio overview --build 2^1..4
```

Note that overviews can not currently be removed and are not automatically updated when the dataset's primary bands are modified.

Information about existing overviews can be printed using the `-ls` option.

```
$ rio overview --ls
```

The block size (tile width and height) used for overviews (internal or external) can be specified by setting the `GDAL_TIFF_OVR_BLOCKSIZE` environment variable to a power-of-two value between 64 and 4096. The default value is 128.

```
$ GDAL_TIFF_OVR_BLOCKSIZE=256 rio overview --build 2^1..4
```

4.13 rasterize

The `rasterize` command rasterizes GeoJSON features into a new or existing raster.

```
$ rio rasterize test.tif --res 0.0167 < input.geojson
```

The resulting file will have an upper left coordinate determined by the bounds of the GeoJSON (in EPSG:4326, which is the default), with a pixel size of approximately 30 arc seconds. Pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in with a default value of 1.

It is possible to rasterize into an existing raster and use an alternative default value:

```
$ rio rasterize existing.tif --default_value 10 < input.geojson
```

It is also possible to rasterize using a template raster, which will be used to determine the transform, dimensions, and coordinate reference system of the output raster:

```
$ rio rasterize test.tif --like tests/data/shade.tif < input.geojson
```

GeoJSON features may be provided using stdin or specified directly as first argument, and dimensions may be provided in place of pixel resolution:

```
$ rio rasterize input.geojson test.tif --dimensions 1024 1024
```

Other options are available, see:

```
$ rio rasterize --help
```

4.14 rm

Invoking the shell's `$ rm <path>` on a dataset can be used to delete a dataset referenced by a file path, but it won't handle deleting side car files. This command is aware of datasets and their sidecar files.

4.15 sample

The `sample` command reads `x`, `y` positions from stdin and writes the dataset values at that position to stdout.

```
$ cat << EOF | rio sample tests/data/RGB.byte.tif
> [220649.99999832606, 2719199.999999095]
> EOF
[18, 25, 14]
```

The output of the transform command (see below) makes good input for `sample`.

4.16 shapes

The `shapes` command extracts and writes features of a specified dataset band out as GeoJSON.

```
$ rio shapes tests/data/shade.tif --bidx 1 --precision 6 --collection > shade.geojson
```

The resulting file looks [like this](#).

Using the `--mask` option you can write out the shapes of a dataset's valid data region.

```
$ rio shapes tests/data/RGB.byte.tif --mask --precision 6 --collection > mask.geojson
```

The output of which looks [like this](#).

Note: `rio shapes` returns line-delimited GeoJSONs by default. Use the `--collection` flag as shown here to return a single GeoJSON feature collection.

4.17 stack

The `stack` command stacks a number of bands from one or more input files into a multiband dataset. Input datasets must be of a kind: same data type, dimensions, etc. The output is cloned from the first input. By default, `stack` will take all bands from each input and write them in same order to the output. Optionally, bands for each input may be specified using the following syntax:

- `--bidx N` takes the Nth band from the input (first band is 1).
- `--bidx M,N,O` takes bands M, N, and O.
- `--bidx M..O` takes bands M-O, inclusive.
- `--bidx ..N` takes all bands up to and including N.
- `--bidx N..` takes all bands from N to the end.

Examples using the Rasterio testing dataset that produce a copy of it.

```
$ rio stack RGB.byte.tif stacked.tif
$ rio stack RGB.byte.tif --bidx 1,2,3 stacked.tif
$ rio stack RGB.byte.tif --bidx 1..3 stacked.tif
$ rio stack RGB.byte.tif --bidx ..2 RGB.byte.tif --bidx 3.. stacked.tif
```

You can use `-rgb` as shorthand for `-co photometric=rgb`.

4.18 transform

The `transform` command reads a JSON array of coordinates, interleaved, and writes another array of transformed coordinates to stdout.

To transform a longitude, latitude point (EPSG:4326 is the default) to another coordinate system with 2 decimal places of output precision, do the following.

```
$ echo "[-78.0, 23.0]" | rio transform - --dst-crs EPSG:32618 --precision 2
[192457.13, 2546667.68]
```

To transform a longitude, latitude bounding box to the coordinate system of a raster dataset, do the following.

```
$ echo "[-78.0, 23.0, -76.0, 25.0]" | rio transform --dst-crs tests/data/RGB.byte.tif -
↳-precision 2
[192457.13, 2546667.68, 399086.97, 2765319.94]
```

4.19 warp

The `warp` command warps (reprojects) a raster based on parameters that can be obtained from a template raster, or input directly. The output is always overwritten.

To copy coordinate reference system, transform, and dimensions from a template raster, do the following:

```
$ rio warp input.tif output.tif --like template.tif
```

You can specify an output coordinate system using a PROJ.4 or EPSG:nnnn string, or a JSON text-encoded PROJ.4 object:

```
$ rio warp input.tif output.tif --dst-crs EPSG:4326
```

```
$ rio warp input.tif output.tif --dst-crs '+proj=longlat +ellps=WGS84 +datum=WGS84'
```

You can also specify dimensions, which will automatically calculate appropriate resolution based on the relationship between the bounds in the target crs and these dimensions:

```
$ rio warp input.tif output.tif --dst-crs EPSG:4326 --dimensions 100 200
```

Or provide output bounds (in source crs) and resolution:

```
$ rio warp input.tif output.tif --dst-crs EPSG:4326 --bounds -78 22 -76 24 --res 0.1
```

Previous command in case of south-up image, `--` escapes the next `-`:

```
$ rio warp input.tif output.tif --dst-crs EPSG:4326 --bounds -78 22 -76 24 --res 0.1 -- -
↳0.1
```

Other options are available, see:

```
$ rio warp --help
```

4.20 Rio Plugins

Rio uses `click-plugins` to provide the ability to create additional subcommands using plugins developed outside rasterio. This is ideal for commands that require additional dependencies beyond those used by rasterio, or that provide functionality beyond the intended scope of rasterio.

For example, `rio-mbtiles` provides a command `rio mbtiles` to export a raster to an MBTiles file.

See [click-plugins](#) for more information on how to build these plugins in general.

To use these plugins with rio, add the commands to the `rasterio.rio_plugins` entry point in your `setup.py` file, as described [here](#) and in `rasterio/rio/main.py`.

See the [plugin registry](#) for a list of available plugins.

4.21 Other commands?

Suggestions for other commands are welcome!

ADVANCED TOPICS

5.1 Using rio-calc

Simple raster data processing on the command line is possible using Rasterio's `rio-calc` command. It uses the [snuggs](#) Numpy S-expression engine. The [snuggs README](#) explains how expressions are written and evaluated in general. This document explains Rasterio-specific details of `rio-calc` and offers some examples.

5.1.1 Expressions

Rio-calc expressions look like

```
(func|operator arg [*args])
```

where `func` may be the name of any function in the module `numpy` or one of the `rio-calc` builtins: `read`, `fillnodata`, or `sieve`; and `operator` may be any of the standard Python arithmetic or logical operators. The arguments may themselves be expressions.

5.1.2 Copying a file

Here's a trivial example of copying a dataset. The expression `(read 1)` evaluates to all bands of the first input dataset, an array with shape `(3, 718, 791)` in this case.

Note: `rio-calc`'s indexes start at 1.

```
$ rio calc "(read 1)" tests/data/RGB.byte.tif out.tif
```

5.1.3 Reversing the band order of a file

The expression `(read i j)` evaluates to the `j`-th band of the `i`-th input dataset. The `asarray` function collects bands read in reverse order into an array with shape `(3, 718, 791)` for output.

```
$ rio calc "(asarray (read 1 3) (read 1 2) (read 1 1))" tests/data/RGB.byte.tif out.tif
```

5.1.4 Stacking bands of multiple files

Bands can be read from multiple input files. This example is another (slower) way to copy a file.

```
$ rio calc "(asarray (read 1 1) (read 2 2) (read 3 3))" \
> tests/data/RGB.byte.tif tests/data/RGB.byte.tif tests/data/RGB.byte.tif \
> out.tif
```

5.1.5 Named datasets

Datasets can be referenced in expressions by name and single bands picked out using the `take` function.

```
$ rio calc "(asarray (take a 3) (take a 2) (take a 1))" \
> --name "a=tests/data/RGB.byte.tif" out.tif
```

The third example, re-done using names, is:

```
$ rio calc "(asarray (take a 1) (take b 2) (take b 3))" \
> --name "a=tests/data/RGB.byte.tif" --name "b=tests/data/RGB.byte.tif" \
> --name "c=tests/data/RGB.byte.tif" out.tif
```

5.1.6 Read and take

The functions `read` and `take` overlap a bit in the previous examples but are rather different. The former involves I/O and the latter does not. You may also `take` from any array, as in this example.

```
$ rio calc "(take (read 1) 1)" tests/data/RGB.byte.tif out.tif
```

5.1.7 Arithmetic operations

Arithmetic operations can be performed as with Numpy. Here is an example of scaling all three bands of a dataset by the same factors.

```
$ rio calc "(+ 2 (* 0.95 (read 1)))" tests/data/RGB.byte.tif out.tif
```

Here is a more complicated example of scaling bands by different factors.

```
$ rio calc "(asarray (+ 2 (* 0.95 (read 1 1))) (+ 3 (* 0.9 (read 1 2))) (+ 4 (* 0.85 (read 1 3))))" tests/data/RGB.byte.tif out.tif
```

5.1.8 Logical operations

Logical operations can be used in conjunction with arithmetic operations. In this example, the output values are 255 wherever the input values are greater than or equal to 40.

```
$ rio calc "(* (>= (read 1) 40) 255)" tests/data/RGB.byte.tif out.tif
```

5.2 Color

5.2.1 Color interpretation

Color interpretation of raster bands can be read from the dataset

```
>>> import rasterio
>>> src = rasterio.open("tests/data/RGB.byte.tif")
>>> src.colorinterp[0]
<ColorInterp.red: 3>
```

GDAL builds the color interpretation based on the driver and creation options. With the GTiff driver, rasters with exactly 3 bands of uint8 type will be RGB, 4 bands of uint8 will be RGBA by default.

Color interpretation can be set when creating a new datasource with the photometric creation option:

```
>>> profile = src.profile
>>> profile['photometric'] = "RGB"
>>> with rasterio.open("/tmp/rgb.tif", 'w', **profile) as dst:
...     dst.write(src.read())
```

or via the colorinterp property when a datasource is opened in update mode:

```
>>> from rasterio.enums import ColorInterp
>>> with rasterio.open('/tmp/rgb.tif', 'r+', **profile) as src:
...     src.colorinterp = [
...         ColorInterp.red, ColorInterp.green, ColorInterp.blue]
```

And the resulting raster will be interpreted as RGB.

```
>>> with rasterio.open("/tmp/rgb.tif") as src2:
...     src2.colorinterp[1]
<ColorInterp.green: 4>
```

5.2.2 Writing colormaps

Mappings from 8-bit (rasterio.uint8) pixel values to RGBA values can be attached to bands using the write_colormap() method.

```
import rasterio

with rasterio.Env():

    with rasterio.open('tests/data/shade.tif') as src:
        shade = src.read(1)
        meta = src.meta

    with rasterio.open('/tmp/colormap.tif', 'w', **meta) as dst:
        dst.write(shade, indexes=1)
        dst.write_colormap(
            1, {
                0: (255, 0, 0, 255),
```

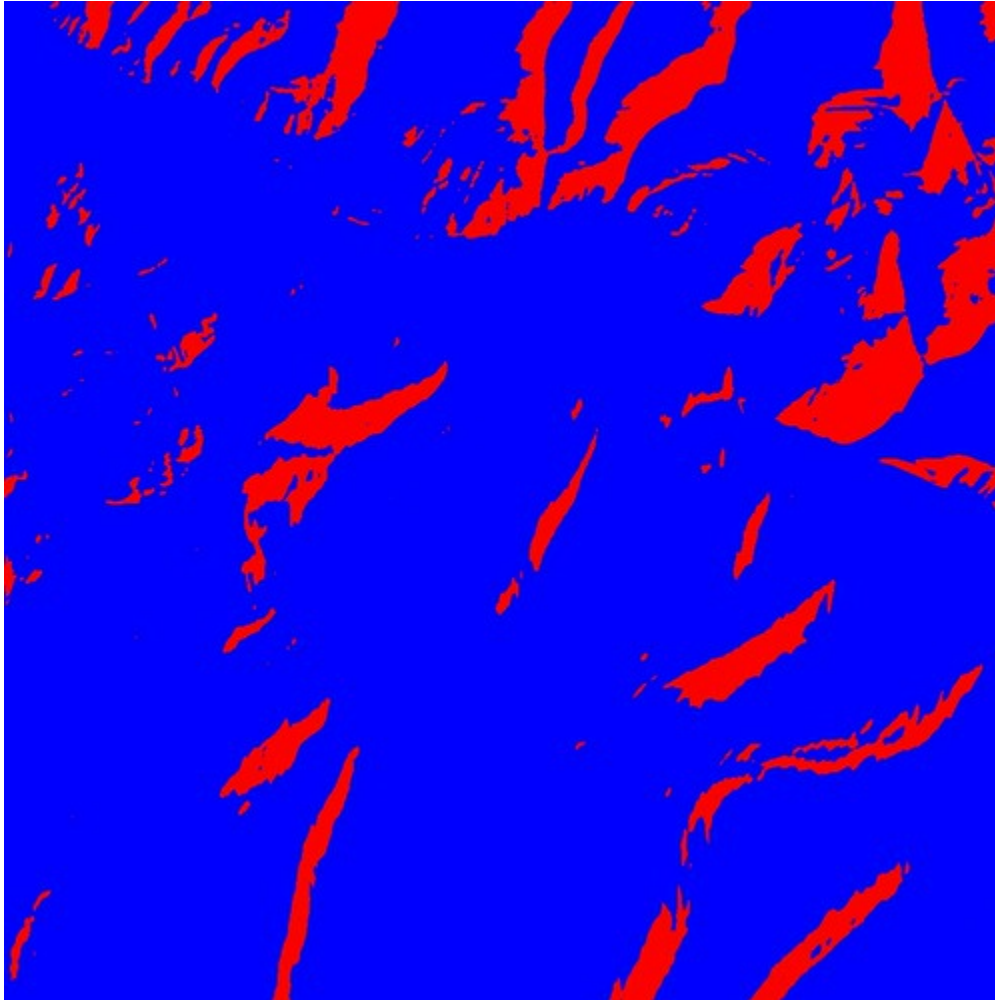
(continues on next page)

(continued from previous page)

```
        255: (0, 0, 255, 255) })
cmap = dst.colormap(1)
# True
assert cmap[0] == (255, 0, 0, 255)
# True
assert cmap[255] == (0, 0, 255, 255)

subprocess.call(['open', '/tmp/colormap.tif'])
```

The program above (on OS X, another viewer is needed with a different OS) yields the image below:



5.2.3 Reading colormaps

As shown above, the `colormap()` returns a dict holding the colormap for the given band index. For TIFF format files, the colormap will have 256 items, and all but two of those would map to (0, 0, 0, 0) in the example above.

5.3 Concurrent processing

Rasterio affords concurrent processing of raster data. Python's global interpreter lock (GIL) is released when calling GDAL's `GDALRasterIO()` function, which means that Python threads can read and write concurrently.

The Numpy library also often releases the GIL, e.g., in applying universal functions to arrays, and this makes it possible to distribute processing of an array across cores of a processor.

This means that it is possible to parallelize tasks that need to be performed for a set of windows/pixels in the raster. Reading, writing and processing can always be done concurrently. But it depends on the hardware and where the bottlenecks are, how much of a speedup can be obtained. In the case that the processing function releases the GIL, multiple threads processing simultaneously can lead to further speedups.

Note: If you wish to do multiprocessing that is not trivially parallelizable across very large images that do not fit in memory, or if you wish to do multiprocessing across multiple machines. You might want to have a look at [dask](#) and in particular this [example](#).

The Cython function below, included in Rasterio's `_example` module, simulates a GIL-releasing CPU-intensive raster processing function. You can also easily create GIL-releasing functions by using [numba](#)

```
# cython: boundscheck=False

import numpy as np

def compute(unsigned char[:, :, :] input):
    """reverses bands inefficiently

    Given input and output uint8 arrays, fakes an CPU-intensive
    computation.
    """
    cdef int I, J, K
    cdef int i, j, k, l
    cdef double val
    I = input.shape[0]
    J = input.shape[1]
    K = input.shape[2]
    output = np.empty((I, J, K), dtype='uint8')
    cdef unsigned char[:, :, :] output_view = output
    with nogil:
        for i in range(I):
            for j in range(J):
                for k in range(K):
                    val = <double>input[i, j, k]
                    for l in range(2000):
                        val += 1.0
                    val -= 2000.0
```

(continues on next page)

(continued from previous page)

```

        output_view[~i, j, k] = <unsigned char>val
    return output

```

Here is the program in `examples/thread_pool_executor.py`. It is set up in such a way that at most 1 thread is reading and at most 1 thread is writing at the same time. Processing is not protected by a lock and can be done by multiple threads simultaneously.

```

"""thread_pool_executor.py

Operate on a raster dataset window-by-window using a ThreadPoolExecutor.

Simulates a CPU-bound thread situation where multiple threads can improve
performance.

With -j 4, the program returns in about 1/4 the time as with -j 1.
"""

import concurrent.futures
import multiprocessing
import threading

import rasterio
from rasterio._example import compute

def main(infile, outfile, num_workers=4):
    """Process infile block-by-block and write to a new file

    The output is the same as the input, but with band order
    reversed.
    """

    with rasterio.open(infile) as src:

        # Create a destination dataset based on source params. The
        # destination will be tiled, and we'll process the tiles
        # concurrently.
        profile = src.profile
        profile.update(blockxsize=128, blockysize=128, tiled=True)

        with rasterio.open(outfile, "w", **src.profile) as dst:
            windows = [window for ij, window in dst.block_windows()]

            # We cannot write to the same file from multiple threads
            # without causing race conditions. To safely read/write
            # from multiple threads, we use a lock to protect the
            # DatasetReader/Writer
            read_lock = threading.Lock()
            write_lock = threading.Lock()

            def process(window):
                with read_lock:

```

(continues on next page)

(continued from previous page)

```

src_array = src.read(window=window)

# The computation can be performed concurrently
result = compute(src_array)

with write_lock:
    dst.write(result, window=window)

# We map the process() function over the list of
# windows.
with concurrent.futures.ThreadPoolExecutor(
    max_workers=num_workers
) as executor:
    executor.map(process, windows)

```

The code above simulates a CPU-intensive calculation that runs faster when spread over multiple cores using `concurrent.futures.ThreadPoolExecutor` compared to the case of one concurrent job (`-j 1`),

```

$ time python examples/thread_pool_executor.py tests/data/RGB.byte.tif /tmp/test.tif -j 1

real    0m4.277s
user    0m4.356s
sys     0m0.184s

```

we get over 3x speed up with four concurrent jobs.

```

$ time python examples/thread_pool_executor.py tests/data/RGB.byte.tif /tmp/test.tif -j 4

real    0m1.251s
user    0m4.402s
sys     0m0.168s

```

If the function that you'd like to map over raster windows doesn't release the GIL, you unfortunately cannot simply replace `ThreadPoolExecutor` with `ProcessPoolExecutor`, the `DatasetReader/DatasetWriter` cannot be shared by multiple processes, which means that each process needs to open the file separately, or you can do all the reading and writing from the main thread, as shown in this next example. This is much less efficient memory wise, however.

```

arrays = [src.read(window=window) for window in windows]

with concurrent.futures.ProcessPoolExecutor(
    max_workers=num_workers
) as executor:
    futures = executor.map(compute, arrays)
    for window, result in zip(windows, futures):
        dst.write(result, window=window)

```

5.4 GDAL Option Configuration

GDAL format drivers and some parts of the library are configurable.

From <https://trac.osgeo.org/gdal/wiki/ConfigOptions>:

ConfigOptions are normally used to alter the default behavior of GDAL and OGR drivers and in some cases the GDAL and OGR core. They are essentially global variables the user can set.

5.4.1 GDAL Example

The following is from GDAL's test suite.

```
gdal.SetConfigOption('GTIFF_FORCE_RGBA', 'YES')
ds = gdal.Open('data/stefan_full_greyalpha.tif')
gdal.SetConfigOption('GTIFF_FORCE_RGBA', None)
```

With GDAL's C or Python API, you call a function once to set a global configuration option before you need it and once again after you're through to unset it.

Downsides of this style of configuration include:

- Options can be configured far from the code they affect.
- There is no API for finding what options are currently set.
- If `gdal.Open()` raises an exception in the code above, the `GTIFF_FORCE_RGBA` option will not be unset.

That code example can be generalized to multiple options and made to recover better from errors.

```
options = {'GTIFF_FORCE_RGBA': 'YES'}
for key, val in options.items():
    gdal.SetConfigOption(key, val)
try:
    ds = gdal.Open('data/stefan_full_greyalpha.tif')
finally:
    for key, val in options.items():
        gdal.SetConfigOption(key, None)
```

This is better, but has a lot of boilerplate. Rasterio uses elements of Python syntax, keyword arguments and the `with` statement, to make this cleaner and easier to use.

5.4.2 Rasterio

```
with rasterio.Env(GTIFF_FORCE_RGBA=True, CPL_DEBUG=True):
    with rasterio.open('data/stefan_full_greyalpha.tif') as dataset:
        # Suite of code accessing dataset ``ds`` follows...
```

The object returned when you call `rasterio.Env` is a context manager. It handles the GDAL configuration for a specific block of code and resets the configuration when the block exits for any reason, success or failure. The Rasterio `with rasterio.Env()` pattern organizes GDAL configuration into single statements and makes its relationship to a block of code clear.

If you want to know what options are configured at any time, you could bind it to a name like so.


```

with rasterio.Env(GTIFF_FORCE_RGBA=True, CPL_DEBUG=True) as env:
    for key, val in env.options.items():
        print(key, val)

# Prints:
# ('GTIFF_FORCE_RGBA', True)
# ('CPL_DEBUG', True)

```

5.4.3 When to use rasterio.Env()

Rasterio code is often without the use of an [Env](#) context block. For instance, you could use `rasterio.open()` directly without explicitly creating an [Env](#). In that case, the `open()` function will initialize a default environment in which to execute the code. Often this default environment is sufficient for most use cases and you only need to create an explicit [Env](#) if you are customizing the default GDAL or format options.

5.5 Advanced Datasets

The analogy of Python file objects influences the design of Rasterio dataset objects. Datasets of a few different kinds exist and the canonical way to obtain one is to call `rasterio.open()` with a path-like object or URI-like identifier, a mode (such as “r” or “w”), and other keyword arguments.

5.5.1 Dataset Identifiers

Datasets in a computer’s filesystem are identified by paths, “file” URLs, or instances of `pathlib.Path`. The following are equivalent.

- `'/path/to/file.tif'`
- `'file:///path/to/file.tif'`
- `pathlib.Path('/path/to/file.tif')`

Datasets within a local zip file are identified using the “zip” scheme from [Apache Commons VFS](#).

- `'zip:///path/to/file.zip!/folder/file.tif'`
- `'zip+file:///path/to/file.zip!/folder/file.tif'`

Note that `!` is the separator between the path of the archive file and the path within the archive file. Also note that this kind of identifier can’t be expressed using `pathlib`.

Similarly, variables of a netCDF dataset can be accessed using “netcdf” scheme identifiers.

```
'netcdf:/path/to/file.nc:variable'
```

Datasets on the web are identified by “http” or “https” URLs such as

- `'https://example.com/file.tif'`
- `'https://landsat-pds.s3.amazonaws.com/L8/139/045/LC81390452014295LGN00/LC81390452014295LGN00_B1.TIF'`

Datasets within a zip file on the web are identified using a “zip+https” scheme and paths separated by `!` as above. For example:

```
'zip+https://example.com/file.tif&p=x&q=y!/folder/file.tif'
```

Datasets on AWS S3 may be identified using “s3” scheme identifiers.

```
's3://landsat-pds/L8/139/045/LC81390452014295LGN00/LC81390452014295LGN00_B1.TIF'
```

Resources in other cloud storage systems will be similarly supported.

5.6 Error Handling

Todo: error enums, context managers, converting GDAL errors to python exceptions

5.6.1 Debugging internal GDAL functions

To get more debugging information from the internal GDAL/PROJ code:

1. Enable the *CPL_DEBUG* config option.

Note: If setting the PROJ_DEBUG environment variable inside a Python script, make sure that it is set before importing rasterio.

```
import os
os.environ["PROJ_DEBUG"] = "2"

import rasterio

with rasterio.Env(CPL_DEBUG=True):
    ...
```

2. Activate logging in *rasterio* with the devel *DEBUG*:

More information available here: <https://docs.python.org/3/howto/logging.html>

Here are examples to get started.

Example - Add handler to the *rasterio* logger:

```
import logging

console_handler = logging.StreamHandler()
formatter = logging.Formatter("%(levelname)s:%(message)s")
console_handler.setFormatter(formatter)
logger = logging.getLogger("rasterio")
logger.addHandler(console_handler)
logger.setLevel(logging.DEBUG)
```

Example - Activate default logging config:

```
import logging

logging.basicConfig(format="%(levelname)s:%(message)s", level=logging.DEBUG)
```

5.7 Vector Features

Rasterio's `features` module provides functions to extract shapes of raster features and to create new features by “burning” shapes into rasters: `shapes()` and `rasterize()`. These functions expose GDAL functions in a general way, using iterators over GeoJSON-like Python objects instead of GIS layers.

5.7.1 Extracting shapes of raster features

Consider the Python logo.



The shapes of the foreground features can be extracted like this:

```
import pprint
import rasterio
from rasterio import features

with rasterio.open('13547682814_f2e459f7a5_o_d.png') as src:
    blue = src.read(3)

# pprint requires that the image dtype must be one of: int16, int32, uint8, uint16,
# float32.
# If your data comes as int8 you can cast your data to an appropriate dtype like this:
# data = data.astype('int16')

mask = blue != 255
shapes = features.shapes(blue, mask=mask)
pprint.pprint(next(shapes))

# Output
# pprint.pprint(next(shapes))
# ({'coordinates': [[(71.0, 6.0),
#                     (71.0, 7.0),
#                     (72.0, 7.0),
#                     (72.0, 6.0),
#                     (71.0, 6.0)]]},
#  'type': 'Polygon'},
# 253)
```

The shapes iterator yields `geometry, value` pairs. The second item is the value of the raster feature corresponding to the shape and the first is its geometry. The coordinates of the geometries in this case are in pixel units with origin at

the upper left of the image. If the source dataset was georeferenced, you would get similarly georeferenced geometries like this:

```
shapes = features.shapes(blue, mask=mask, transform=src.transform)
```

5.7.2 Burning shapes into a raster

To go the other direction, use `rasterize()` to burn values into the pixels intersecting with geometries.

```
image = features.rasterize(
    ((g, 255) for g, v in shapes),
    out_shape=src.shape)
```

By default, only pixels whose center is within the polygon or that are selected by Bresenham’s line algorithm will be burned in. You can specify `all_touched=True` to burn in all pixels touched by the geometry. The geometries will be rasterized by the “painter’s algorithm” - geometries are handled in order and later geometries will overwrite earlier values.

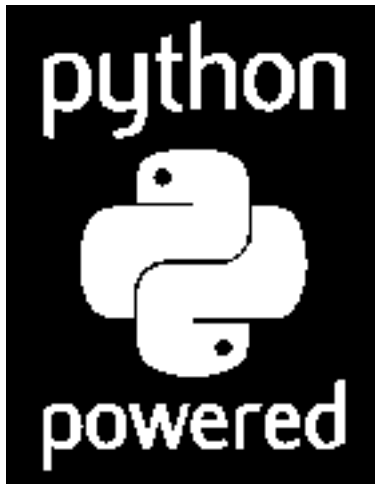
Again, to burn in georeferenced shapes, pass an appropriate transform for the image to be created.

```
image = features.rasterize(
    ((g, 255) for g, v in shapes),
    out_shape=src.shape,
    transform=src.transform)
```

The values for the input shapes are replaced with 255 in a generator expression. Areas not covered by input geometries are replaced with an optional fill value, which defaults to 0. The resulting image, written to disk like this,

```
with rasterio.open(
    '/tmp/rasterized-results.tif', 'w',
    driver='GTiff',
    dtype=rasterio.uint8,
    count=1,
    width=src.width,
    height=src.height) as dst:
    dst.write(image, indexes=1)
```

has a black background and white foreground features.



5.8 Filling nodata areas

Todo: `fillnodata()`

5.9 Georeferencing

There are two parts to the georeferencing of raster datasets: the definition of the local, regional, or global system in which a raster's pixels are located; and the parameters by which pixel coordinates are transformed into coordinates in that system.

5.9.1 Coordinate Reference System

The coordinate reference system of a dataset is accessed from its `DatasetReader.crs` attribute.

```
>>> import rasterio
>>> src = rasterio.open('tests/data/RGB.byte.tif')
>>> src.crs
CRS({'init': 'epsg:32618'})
```

Rasterio follows pyproj and uses PROJ.4 syntax in dict form as its native CRS syntax. If you want a WKT representation of the CRS, see: `CRS.to_wkt()`:

```
>>> src.crs.to_wkt()
'PROJCS["WGS 84 / UTM zone 18N",GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS 84",
↳ 6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],PRIMEM[
↳ "Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.0174532925199433,AUTHORITY[
↳ "EPSG","9122"]],AUTHORITY["EPSG","4326"]],PROJECTION["Transverse_Mercator"],PARAMETER[
↳ "latitude_of_origin",0],PARAMETER["central_meridian",-75],PARAMETER["scale_factor",0.
↳ 9996],PARAMETER["false_easting",500000],PARAMETER["false_northing",0],UNIT["metre",1,
↳ AUTHORITY["EPSG","9001"]],AXIS["Easting",EAST],AXIS["Northing",NORTH],AUTHORITY["EPSG",
↳ "32618"]]'
```

When opening a new file for writing, you may also use a CRS string as an argument.

```
>>> profile = {'driver': 'GTiff', 'height': 100, 'width': 100, 'count': 1, 'dtype': '
↳ rasterio.uint8'}
>>> with rasterio.open('/tmp/foo.tif', 'w', crs='EPSG:3857', **profile) as dst:
...     pass # write data to this Web Mercator projection dataset.
```

5.9.2 Coordinate Transformation

This section describes the three primary kinds of georeferencing metadata supported by rasterio.

Affine

A dataset's pixel coordinate system has its origin at the “upper left” (imagine it displayed on your screen). Column index increases to the right, and row index increases downward. The mapping of these coordinates to “world” coordinates in the dataset's reference system is typically done with an affine transformation matrix.

```
>>> src.transform
Affine(300.0379266750948, 0.0, 101985.0,
       0.0, -300.041782729805, 2826915.0)
```

The `Affine` object is a named tuple with elements `a`, `b`, `c`, `d`, `e`, `f` corresponding to the elements in the matrix equation below, in which a pixel's image coordinates are `x`, `y` and its world coordinates are `x'`, `y'`:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The `Affine` class has some useful properties and methods described at <https://github.com/sgillies/affine>.

Some datasets may not have an affine transformation matrix, but are still georeferenced.

Ground Control Points

A ground control point (GCP) is the mapping of a dataset's row and pixel coordinate to a single world `x`, `y`, and optionally `z` coordinate. Typically a dataset will have multiple GCPs distributed across the image. Rasterio can calculate an affine transformation matrix from a collection of GCPs using the `rasterio.transform.from_gcps()` method. Alternatively GCP interpolation can also be used for coordinate transforms.

Rational Polynomial Coefficients

A dataset may also be georeferenced with a set of rational polynomial coefficients (RPCs) which can be used to compute pixel coordinates from `x`, `y`, and `z` coordinates. The RPCs are an application of the Rigorous Projection Model which uses four sets of 20 term cubic polynomials and several normalizing parameters to establish a relationship between image and world coordinates. RPCs are defined with image coordinates in pixel units and world coordinates in decimal degrees of longitude and latitude and height above the WGS84 ellipsoid (EPSG:4326).

RPCs are usually provided by the dataset provider and are only well behaved over the extent of the image. Additionally, accurate height values are required for the best results. Datasets with low terrain variation may use an average height over the extent of the image, while datasets with higher terrain variation should use a digital elevation model to sample height values. The coordinate transformation from world to pixel coordinates is exact while the reverse is not, and must be computed iteratively. For more details on coordinate transformations using RPCs see `GDALCreateRPCTransformerV2()`.

5.10 Options

GDAL's format drivers have many [configuration options](#). These options come in two flavors:

- **Configuration options** are used to alter the default behavior of GDAL and OGR and are generally treated as global environment variables by GDAL. These are set through a `rasterio.Env` context block in Python.
- **Creation options** are passed into the driver at dataset creation time as keyword arguments to `rasterio.open(mode='w')`.

5.10.1 Configuration Options

GDAL options are typically set as environment variables. While environment variables will influence the behavior of `rasterio`, we highly recommended avoiding them in favor of defining behavior programmatically.

The preferred way to set options for `rasterio` is via `rasterio.Env`. Options set on entering the context are deleted on exit.

```
import rasterio

with rasterio.Env(GDAL_TIFF_INTERNAL_MASK=True):
    # GeoTIFFs written here will have internal masks, not the
    # .msk sidecars.
    # ...

# Option is gone and the default (False) returns.
```

Use native Python forms (`True` and `False`) for boolean options. `Rasterio` will convert them GDAL's internal forms.

See the [configuration options](#) page for a complete list of available options.

5.10.2 Creation options

Each format has it's own set of driver-specific creation options that can be used to fine tune the output rasters. For details on a particular driver, see the [formats list](#).

For the purposes of this document, we will focus on the [GeoTIFF creation options](#). Some of the common GeoTIFF creation options include:

- `TILED`, `BLOCKXSIZE`, and `BLOCKYSIZE` to define the internal tiling
- `COMPRESS` to define the compression method
- `PHOTOMETRIC` to define the band's color interpretation

To specify these creation options in python code, you pass them as keyword arguments to the `rasterio.open()` command in write mode.

```
with rasterio.open("output.tif", 'w', **src.meta, compress="JPEG",
                  tiled=True, blockxsize=256, blockysize=256,
                  photometric="YCBCR") as dataset:
    # Write data to the dataset.
```

Note: The GeoTIFF format requires that *blockxsize* and *blockysize* be multiples of 16.

On the command line, rio commands will accept multiple `--co` options.

```
$ rio copy source.tif dest.tif --co tiled=true
```

These keyword arguments may be lowercase or uppercase, as you prefer.

Attention: Some options may at a glance appear to be boolean, but are not. The GeoTIFF format's BIGTIFF option is one of these. The value must be YES, NO, IF_NEEDED, or IF_SAFER.

Note: Some *configuration* options also have an effect on driver behavior at creation time.

5.11 Interoperability

5.11.1 Image processing software

Some python image processing software packages organize arrays differently than rasterio. The interpretation of a 3-dimension array read from rasterio is:

```
(bands, rows, columns)
```

while image processing software like `scikit-image`, `pillow` and `matplotlib` are generally ordered:

```
(rows, columns, bands)
```

The number of rows defines the dataset's height, the columns are the dataset's width.

Numpy provides a way to efficiently swap the axis order and you can use the following reshape functions to convert between raster and image axis order:

```
>>> import rasterio
>>> from rasterio.plot import reshape_as_raster, reshape_as_image

>>> raster = rasterio.open("tests/data/RGB.byte.tif").read()
>>> raster.shape
(3, 718, 791)

>>> image = reshape_as_image(raster)
>>> image.shape
(718, 791, 3)

>>> raster2 = reshape_as_raster(image)
>>> raster2.shape
(3, 718, 791)
```


5.12 Masking a raster using a shapefile

Using `rasterio` with `fiona`, it is simple to open a shapefile, read geometries, and mask out regions of a raster that are outside the polygons defined in the shapefile.

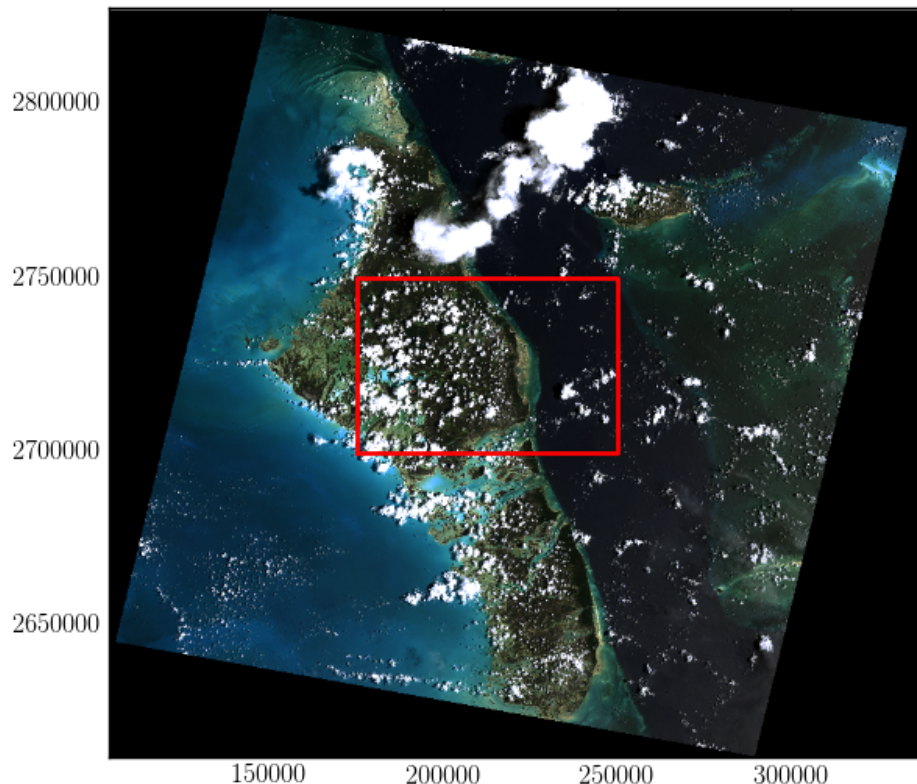
```
import fiona
import rasterio
import rasterio.mask

with fiona.open("tests/data/box.shp", "r") as shapefile:
    shapes = [feature["geometry"] for feature in shapefile]
```

This shapefile contains a single polygon, a box near the center of the raster, so in this case, our list of features is one element long.

```
with rasterio.open("tests/data/RGB.byte.tif") as src:
    out_image, out_transform = rasterio.mask.mask(src, shapes, crop=True)
    out_meta = src.meta
```

Using `matplotlib.pyplot.plot()` and `matplotlib.pyplot.imshow()`, we can see the region defined by the shapefile in red overlaid on the original raster.

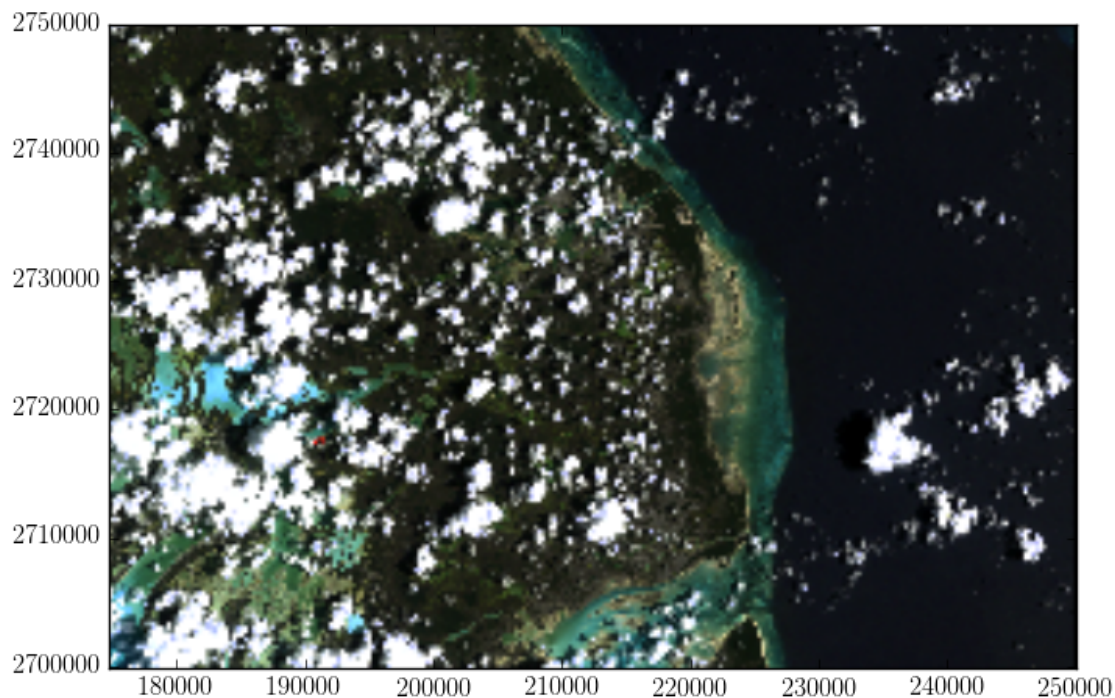


Applying the features in the shapefile as a mask on the raster sets all pixels outside of the features to be zero. Since `crop=True` in this example, the extent of the raster is also set to be the extent of the features in the shapefile. We can

then use the updated spatial transform and raster height and width to write the masked raster to a new file.

```
out_meta.update({"driver": "GTiff",
                 "height": out_image.shape[1],
                 "width": out_image.shape[2],
                 "transform": out_transform})

with rasterio.open("RGB.byte.masked.tif", "w", **out_meta) as dest:
    dest.write(out_image)
```



5.13 Nodata Masks

Nodata masks allow you to identify regions of valid data values. In using Rasterio, you'll encounter two different kinds of masks.

One is the the valid data mask from GDAL, an unsigned byte array with the same number of rows and columns as the dataset in which non-zero elements (typically 255) indicate that the corresponding data elements are valid. Other elements are invalid, or *nodata* elements.

The other kind of mask is a `numpy.ma.MaskedArray` which has the inverse sense: *True* values in a masked array's mask indicate that the corresponding data elements are invalid. With care, you can safely navigate convert between the two mask types.

Consider Rasterio's RGB.byte.tif test dataset. It has 718 rows and 791 columns of pixels. Each pixel has 3 8-bit (uint8) channels or bands. It has a trapezoid of image data within a rectangular background of 0,0,0 value pixels.



Metadata in the dataset declares that values of 0 will be interpreted as invalid data or *nodata* pixels. In, e.g., merging the image with adjacent scenes, we'd like to ignore the nodata pixels and have only valid image data in our final mosaic.

Let's look at the two kinds of masks and their inverse relationship in the context of RGB.byte.tif.

```
>>> import rasterio
>>> src = rasterio.open("tests/data/RGB.byte.tif")
>>> src.shape
(718, 791)
>>> src.count
3
>>> src.dtypes
('uint8', 'uint8', 'uint8')
>>> src.nodatavals
(0.0, 0.0, 0.0)
```

(continues on next page)

(continued from previous page)

```
>>> src.nodata
0.0
```

5.13.1 Reading dataset masks

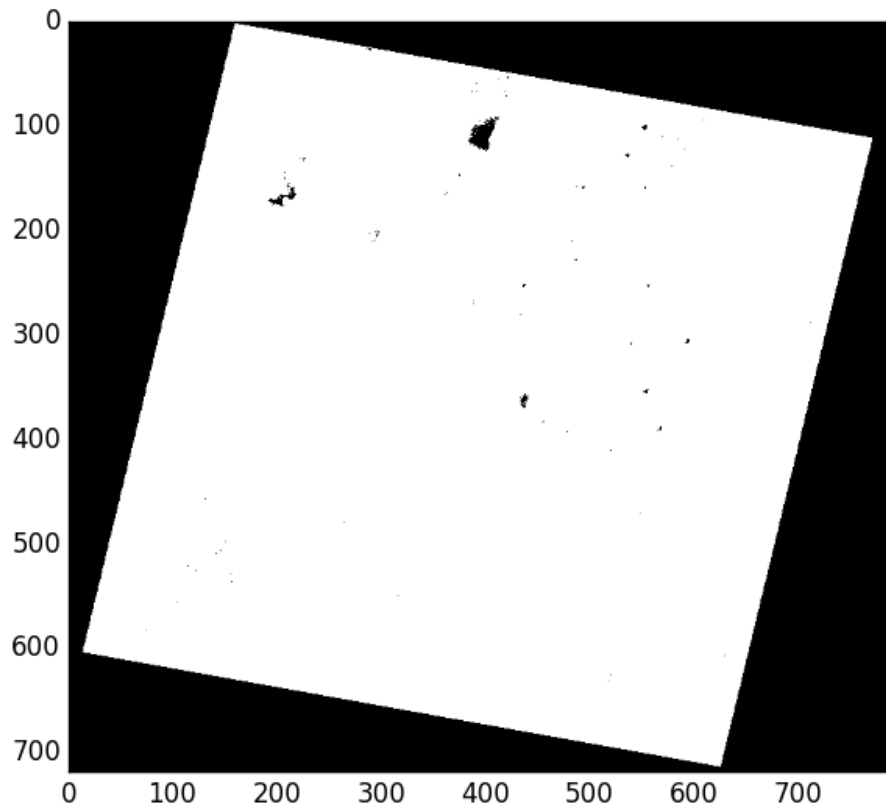
For every band of a dataset there is a mask. These masks can be had as arrays using the dataset's `read_masks()` method. Below, `msk` is the valid data mask corresponding to the first dataset band.

```
>>> msk = src.read_masks(1)
>>> msk.shape
(718, 791)
>>> msk
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

This 2D array is a valid data mask in the sense of [GDAL RFC 15](#). The `0` values in its corners represent *nodata* regions. Zooming in on the interior of the mask array shows the `255` values that indicate *valid data* regions.

```
>>> msk[200:205, 200:205]
array([[255, 255, 255, 255, 255],
       [255, 255, 255, 255, 255],
       [255, 255, 255, 255, 255],
       [255, 255, 255, 255, 255],
       [255, 255, 255, 255, 255]], dtype=uint8)
```

Displayed using `matplotlib.pyplot.imshow()`, the mask looks like this:



Wait, what are these 0 values in the mask interior? This is an example of a problem inherent in 8-bit raster data: lack of dynamic range. The dataset creator has said that 0 values represent missing data (see the `nodatavals` property in the first code block of this document), but some of the valid data have values so low they've been rounded during processing to zero. This can happen in scaling 16-bit data to 8 bits. There's no magic nodata value bullet for this. Using 16 bits per band helps, but you really have to be careful with 8-bit per band datasets and their nodata values.

5.13.2 Writing masks

Writing a mask that applies to all dataset bands is just as straightforward: pass an ndarray with `True` (or values that evaluate to `True` to indicate valid data and `False` to indicate no data to `write_mask()`). Consider a copy of the test data opened in "r+" (update) mode.

```
>>> import shutil
>>> import rasterio

>>> tmp = shutil.copy("tests/data/RGB.byte.tif", "/tmp/RGB.byte.tif")
>>> src = rasterio.open(tmp, mode="r+")
```

To mark that all pixels of all bands are valid (i.e., to override nodata metadata values that can't be unset), you'd do this.

```
>>> src.write_mask(True)
>>> src.read_masks(1).all()
True
```


No data have been altered, nor have the dataset's nodata values been changed. A new band has been added to the dataset to store the valid data mask. By default it is saved to a "sidecar" GeoTIFF alongside the dataset file. When such a .msk GeoTIFF exists, Rasterio will ignore the nodata metadata values and return mask arrays based on the .msk file.

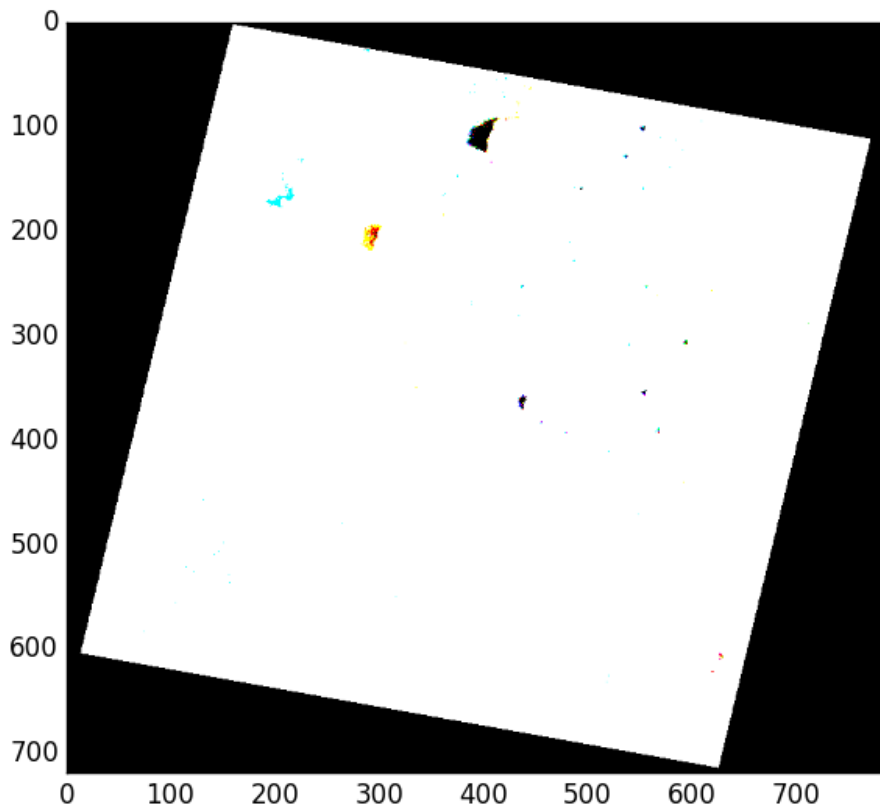
```
$ ls -l copy.tif*
-rw-r--r--@ 1 sean  staff  1713704 Mar 24 14:19 copy.tif
-rw-r--r--  1 sean  staff    916 Mar 24 14:25 copy.tif.msk
```

Can Rasterio help fix buggy nodata masks like the ones in RGB.byte.tif? It certainly can. Consider a fresh copy of that file.

```
>>> src.close()
>>> tmp = shutil.copy("tests/data/RGB.byte.tif", "/tmp/RGB.byte.tif")
>>> src = rasterio.open(tmp, mode="r+")
```

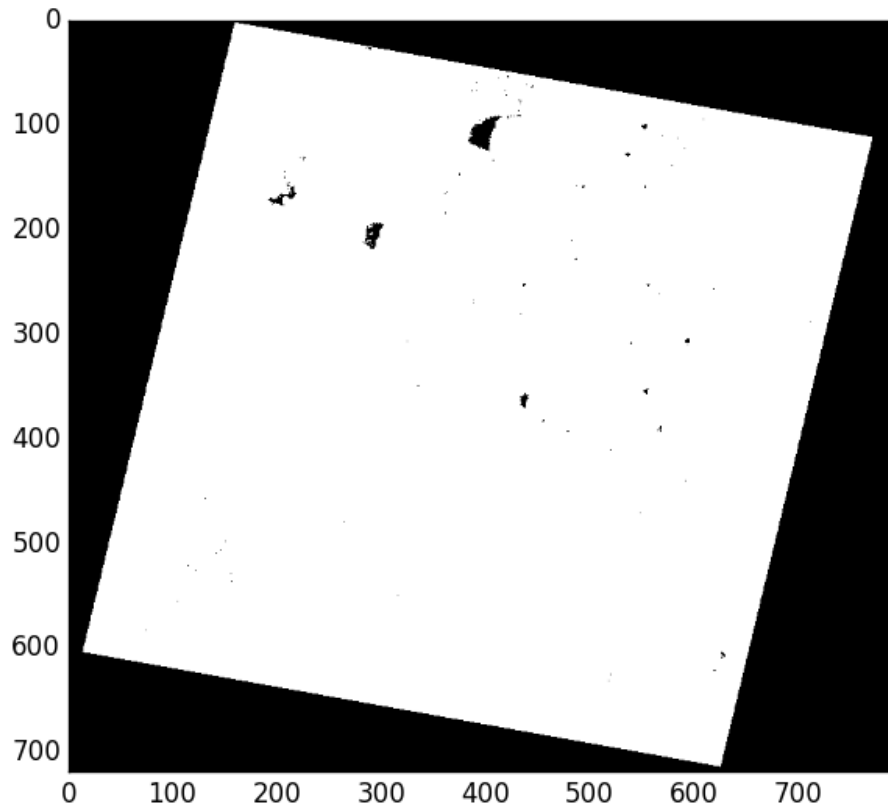
This time we'll read all 3 band masks (based on the nodata values, not a .msk GeoTIFF) and show them as an RGB image (with the help of `numpy.dstack()`):

```
>>> msk = src.read_masks()
>>> show(np.dstack(msk))
```



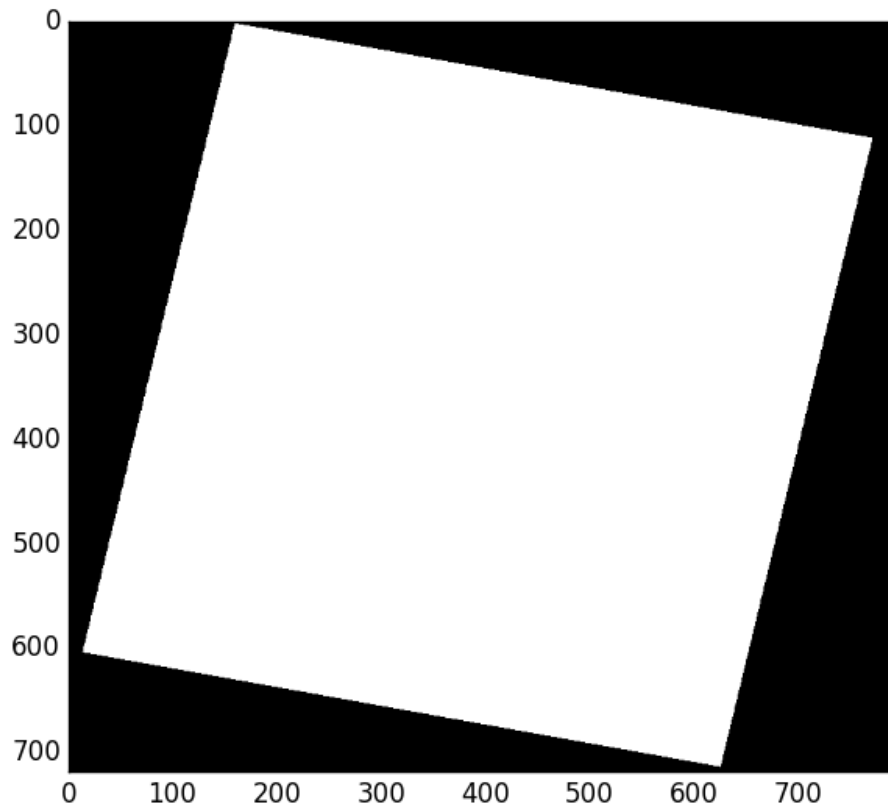
Colored regions appear where valid data pixels don't quite coincide. This is, again, an artifact of scaling data down to 8 bits per band. We'll begin by constructing a new mask array from the logical conjunction of the three band masks we've read.

```
>>> new_msk = (msk[0] & msk[1] & msk[2])  
>>> show(new_msk)
```



Now we'll use `sieve()` to shake out the small buggy regions of the mask. I've found the right value for the `size` argument empirically.

```
>>> from rasterio.features import sieve  
>>> sieved_msk = sieve(new_msk, size=800)  
>>> show(sieved_msk)
```



Last thing to do is write that sieved mask back to the dataset.

```
>>> src.write_mask(sieved_msk)
>>> src.close()
```

The result is a properly masked dataset that allows some 0 value pixels to be considered valid.

5.13.3 Numpy masked arrays

If you want, you can read dataset bands as a `numpy.ma.MaskedArray`.

```
>>> src = rasterio.open("tests/data/RGB.byte.tif")
>>> blue = src.read(1, masked=True)
>>> blue.mask
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True]], dtype=bool)
```

As mentioned earlier, this mask is the inverse of the GDAL band mask. To get a mask conforming to GDAL RFC 15, do this:


```
>>> msk = (~blue.mask * 255).astype('uint8')
```

You can rely on this Rasterio identity for any integer value N.

```
>>> N = 1
>>> (~src.read(N, masked=True).mask * 255 == src.read_masks(N)).all()
True
```

5.13.4 Dataset masks

Sometimes a per-band mask is not appropriate. In this case you can either construct a mask out of the component bands (or other auxillary data) manually *or* use the Rasterio dataset's `dataset_mask()` function. This returns a 2D array with a GDAL-style mask determined by the following criteria, in order of precedence:

1. If a `.msk` file, dataset-wide alpha or internal mask exists, it will be used as the dataset mask.
2. If a 4-band RGBA with a shadow nodata value, band 4 will be used as the dataset mask.
3. If a nodata value exists, use the binary OR (!) of the band masks
4. If no nodata value exists, return a mask filled with all valid data (255)

Note that this differs from `read_masks` and GDAL RFC15 in that it applies per-dataset, not per-band.

5.13.5 Nodata representations in raster files

The storage and representation of nodata differs depending on the data format and configuration options. While Rasterio provides an abstraction for those details when reading, it's often important to understand the differences when creating, manipulating and writing raster data.

- **Nodata values:** the nodata value is used to define which pixels should be masked.
- **Alpha band:** with RGB imagery, an additional 4th band (containing a GDAL-style 8-bit mask) is sometimes provided to explicitly define the mask.
- **Internal mask band:** GDAL provides the ability to store an additional boolean 1-bit mask that is stored internally to the dataset. This option relies on a GDAL environment with `GDAL_TIFF_INTERNAL_MASK=True`. Otherwise the mask will be written externally.
- **External mask band:** Same as above but the mask band is stored in a sidecar `.msk` file (default).

5.14 In-Memory Files

Other sections of this documentation have explained how Rasterio can access data stored in existing files on disk written by other programs or write files to be used by other GIS programs. Filenames have been the typical inputs and files on disk have been the typical outputs.

```
with rasterio.open('example.tif') as dataset:
    data_array = dataset.read()
```

There are different options for Python programs that have streams of bytes, e.g., from a network socket, as their input or output instead of filenames. One is the use of a temporary file on disk.

```
import tempfile

with tempfile.NamedTemporaryFile() as tmpfile:
    tmpfile.write(data)
    with rasterio.open(tmpfile.name) as dataset:
        data_array = dataset.read()
```

Another is Rasterio's `MemoryFile`, an abstraction for objects in GDAL's in-memory filesystem.

5.14.1 MemoryFile: BytesIO meets NamedTemporaryFile

The `MemoryFile` class behaves a bit like `BytesIO` and `NamedTemporaryFile()`. A GeoTIFF file in a sequence of data bytes can be opened in memory as shown below.

```
from rasterio.io import MemoryFile

with MemoryFile(data) as memfile:
    with memfile.open() as dataset:
        data_array = dataset.read()
```

This code can be several times faster than the code using `NamedTemporaryFile()` at roughly double the price in memory.

5.14.2 Writing MemoryFiles

Incremental writes to an empty `MemoryFile` are also possible.

```
with MemoryFile() as memfile:
    while True:
        data = f.read(8192)  # ``f`` is an input stream.
        if not data:
            break
        memfile.write(data)
    with memfile.open() as dataset:
        data_array = dataset.read()
```

These two modes are incompatible: a `MemoryFile` initialized with a sequence of bytes cannot be extended.

An empty `MemoryFile` can also be written to using dataset API methods.

```
with MemoryFile() as memfile:
    with memfile.open(driver='GTiff', count=3, ...) as dataset:
        dataset.write(data_array)
```

5.14.3 Reading MemoryFiles

Like `BytesIO`, `MemoryFile` implements the Python file protocol and provides `read()`, `seek()`, and `tell()` methods. Instances are thus suitable as arguments for methods like `requests.post()`.

```
with MemoryFile() as memfile:
    with memfile.open(driver='GTiff', count=3, ...) as dataset:
        dataset.write(data_array)

requests.post('https://example.com/upload', data=memfile)
```

5.15 Migrating to Rasterio 1.0

5.15.1 `affine.Affine()` vs. GDAL-style geotransforms

One of the biggest API changes on the road to Rasterio 1.0 is the full deprecation of GDAL-style geotransforms in favor of the `affine` library. For reference, an `affine.Affine()` looks like:

```
affine.Affine(a, b, c,
              d, e, f)
```

and a GDAL geotransform looks like:

```
(c, a, b, f, d, e)
```

Fundamentally these two constructs provide the same information, but the `Affine()` object is more useful.

Here's a history of this feature:

1. Originally, functions with a `transform` argument expected a GDAL geotransform.
2. The introduction of the `affine` library involved creating a temporary `affine` argument for `rasterio.open()` and a `src.affine` property. Users could pass an `Affine()` to `affine` or `transform`, but a GDAL geotransform passed to `transform` would issue a deprecation warning.
3. `src.transform` remained a GDAL geotransform, but issued a warning. Users were pointed to `src.affine` during the transition phase.
4. Since the above changes, several functions have been added to Rasterio that accept a `transform` argument. Rather than add an `affine` argument to each, the `transform` argument could be either an `Affine()` object or a GDAL geotransform, the latter issuing the same deprecation warning.

The original plan was to remove the `affine` argument + property, and assume that the object passed to `transform` is an `Affine()`. However, after [further discussion](#) it was determined that since `Affine()` and GDAL geotransforms are both 6 element tuples users may experience unexplained errors and outputs, so an exception is raised instead to better highlight the error.

Before 1.0b1:

- `rasterio.open()` will still accept `affine` and `transform`, but the former now issues a deprecation warning and the latter raises an exception if it does not receive an `Affine()`.
- If `rasterio.open()` receives both `affine` and `transform` a warning is issued and `transform` is used.
- `src.affine` remains but issues a deprecation warning.
- `src.transform` returns an `Affine()`.

- All other Rasterio functions with a `transform` argument now raise an exception if they receive a GDAL geotransform.

Tickets

- [#86](#) - Announcing the plan to switch from GDAL geotransforms to `Affine()`.
- [#763](#) - Implementation of the migration and some further discussion.
Beginning in 1.0b1:
 - In `rasterio.open` “affine” will no longer be an alias for the transform keyword argument.
 - Dataset objects will no longer have an affine property.
 - The transform keyword argument and property is always an instance of the `Affine` class.

I/O Operations

Methods related to reading band data and dataset masks have changed in 1.0.

Beginning with version 1.0b1, there is no longer a `read_mask` method, only `read_masks`. Datasets may be opened in read-write “w+” mode when their formats allow and a warning will be raised when band data or masks are read from datasets opened in “w” mode.

Beginning with 1.0.0, the “w” mode will become write-only and reading data or masks from datasets opened in “w” will be prohibited.

5.15.2 Deprecated: `rasterio.drivers()`

Previously users could register GDAL’s drivers and open a datasource with:

```
import rasterio

with rasterio.drivers():

    with rasterio.open('tests/data/RGB.byte.tif') as src:
        pass
```

but Rasterio 1.0 contains more interactions with GDAL’s environment, so `rasterio.drivers()` has been replaced with:

```
import rasterio
import rasterio.env

with rasterio.Env():

    with rasterio.open('tests/data/RGB.byte.tif') as src:
        pass
```

Tickets

- [#665](#) - Deprecation of `rasterio.drivers()` and introduction of `rasterio.Env()`.

Removed: `src.read_band()`

The `read_band()` method has been replaced by `read()`, which allows for faster I/O and reading multiple bands into a single `numpy.ndarray`.

For example:

```
import numpy as np
import rasterio

with rasterio.open('tests/data/RGB.byte.tif') as src:
    data = np.array(map(src.read_band, (1, 2, 3)))
    band1 = src.read_band(1)
```

is now:

```
import rasterio

with rasterio.open('tests/data/RGB.byte.tif') as src:
    data = src.read((1, 2, 3))
    band1 = src.read(1)
```

Tickets

- [# 83](#) - Introduction of `src.read()`.
- [#96](#), [#284](#) - Deprecation of `src.read_band()`.

Removed: `src.read_mask()`

The `src.read_mask()` method produced a single mask for the entire datasource, but could not handle producing a single mask per band, so it was deprecated in favor of `src.read_masks()`, although it has no direct replacement.

Tickets

- [#284](#) - Deprecation of `src.read_mask()`.

5.15.3 Moved: Functions for working with dataset windows

Several functions in the top level `rasterio` namespace for working with dataset windows have been moved to `rasterio.windows.*`:

- `rasterio.get_data_window()`
- `rasterio.window_union()`
- `rasterio.window_intersection()`
- `rasterio.windows_intersect()`

Tickets

- [#609](#) - Introduction of `rasterio.windows`.

5.15.4 Moved: `rasterio.tool`

This module has been removed completely and its contents have been moved to several different locations:

```
rasterio.tool.show()      -> rasterio.plot.show()
rasterio.tool.show_hist() -> rasterio.plot.show_hist()
rasterio.tool.stats()     -> rasterio.rio.insp.stats()
rasterio.tool.main()      -> rasterio.rio.insp.main()
```

Tickets

- [#609](#) - Deprecation of `rasterio.tool`.

5.15.5 Moved: `rasterio.tools`

This module has been removed completely and its contents have been moved to several different locations:

```
rasterio.tools.mask.mask()  -> rasterio.mask.mask()
rasterio.tools.merge.merge() -> rasterio.merge.merge()
```

Tickets

- [#609](#) - Deprecation of `rasterio.tools`.

5.15.6 Removed: `rasterio.warp.RESAMPLING`

This enum has been replaced by `rasterio.warp.Resampling`.

5.15.7 Removed: `dataset's ul()` method

This method has been replaced by the `xy()` method.

5.15.8 Signature Changes

For both `rasterio.features.sieve()` and `rasterio.features.rasterize()` the output argument has been replaced with `out`. Previously the use of `output` issued a deprecation warning.

5.15.9 Deprecation of dataset property `set_*` and `get_*` methods

Methods `get_crs`, `set_crs`, `set_nodatavals`, `set_descriptions`, `set_units`, and `set_gcps` are deprecated and will be removed in version 1.0. They have been replaced by fully settable dataset properties `crs`, `nodatavals`, `descriptions`, `units`, and `gcps`.

In the cases of units and descriptions, `set_band_unit` and `set_band_description` methods remain to support the `rio-edit-info` command.

5.15.10 Creation Options

Rasterio no longer saves dataset creation options to the metadata of created datasets and will ignore such metadata starting in version 1.0. Users may opt in to this by setting `RIO_IGNORE_CREATION_KWDS=TRUE` in their environments.

5.16 Overviews

Overviews are reduced resolution versions of your dataset that can speed up rendering when you don't need full resolution. By precomputing the upsampled pixels, rendering can be significantly faster when zoomed out.

Overviews can be stored internally or externally, depending on the file format.

In some cases we may want to make a copy of the test data to avoid altering the original.

```
>>> import shutil
>>> path = shutil.copy('tests/data/RGB.byte.tif', '/tmp/RGB.byte.tif')
```

We must specify the zoom factors for which to build overviews. Commonly these are exponents of 2

```
>>> factors = [2, 4, 8, 16]
```

To control the visual quality of the overviews, the 'nearest', 'cubic', 'average', 'mode', and 'gauss' resampling algorithms are available. These are available through the `Resampling` enum

```
>>> from rasterio.enums import Resampling
```

Creating overviews requires opening a dataset in `r+` mode, which gives us access to update the data in place. By convention we also add a tag in the `rio_overview` namespace so that readers can determine what resampling method was used.

```
>>> import rasterio
>>> dst = rasterio.open(path, 'r+')
>>> dst.build_overviews(factors, Resampling.average)
>>> dst.update_tags(ns='rio_overview', resampling='average')
>>> dst.close()
```

We can read the updated dataset and confirm that the overviews are present

```
>>> src = rasterio.open(path, 'r')
>>> [src.overviews(i) for i in src.indexes]
[[2, 4, 8, 16], [2, 4, 8, 16], [2, 4, 8, 16]]
>>> src.tags(ns='rio_overview').get('resampling')
'average'
```

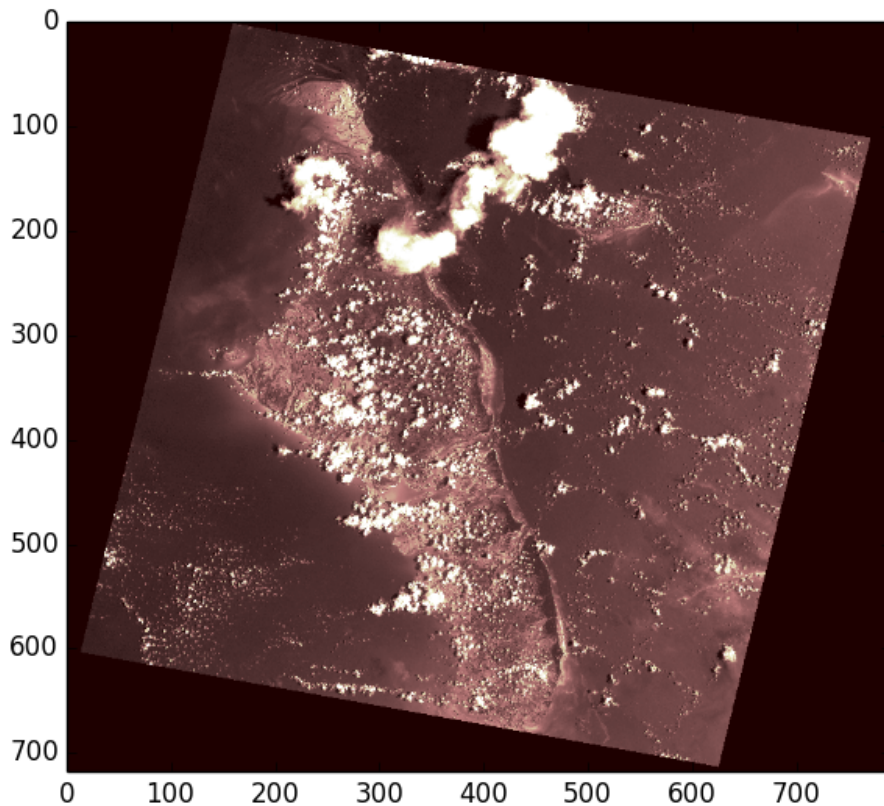
And to leverage the overviews, we can perform a decimated read at a reduced resolution which should allow libgdal to read directly from the overviews rather than compute them on-the-fly.

```
>>> src.read().shape
(3, 718, 791)
>>> src.read(out_shape=(3, int(src.height / 4), int(src.width / 4))).shape
(3, 179, 197)
```

5.17 Plotting

Rasterio reads raster data into numpy arrays so plotting a single band as two dimensional data can be accomplished directly with pyplot.

```
>>> import rasterio
>>> from matplotlib import pyplot
>>> src = rasterio.open("tests/data/RGB.byte.tif")
>>> pyplot.imshow(src.read(1), cmap='pink')
<matplotlib.image.AxesImage object at 0x...>
>>> pyplot.show()
```



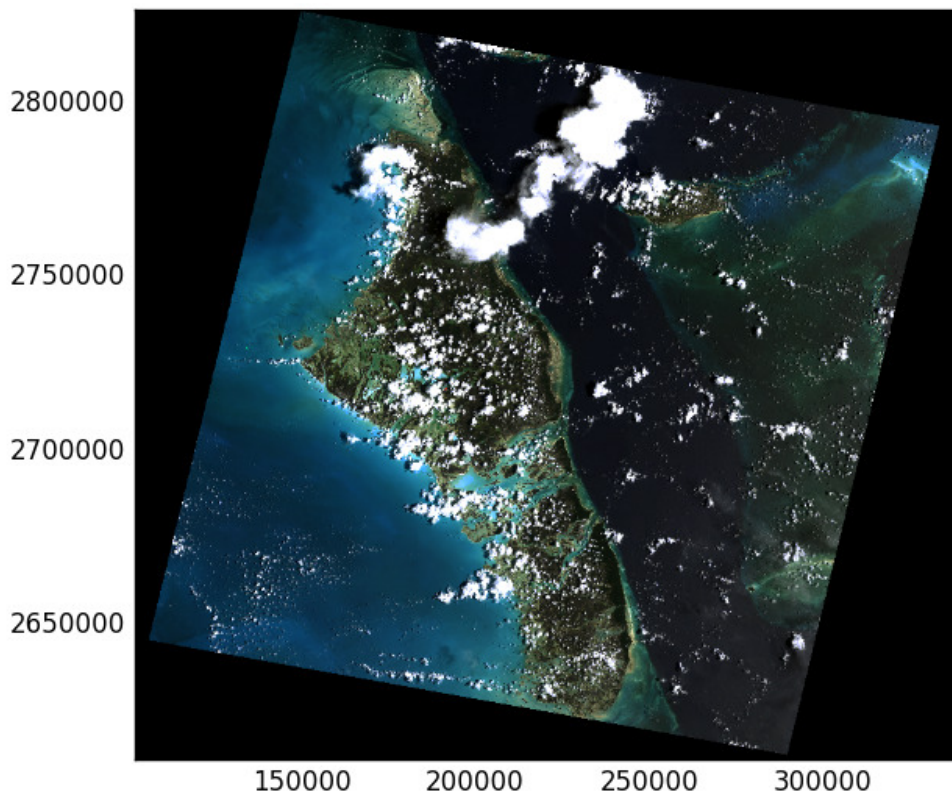
Rasterio also provides `rasterio.plot.show()` to perform common tasks such as displaying multi-band images as RGB and labeling the axes with proper geo-referenced extents.

The first argument to `show()` represent the data source to be plotted. This can be one of

- A dataset object opened in 'r' mode
- A single band of a source, represented by a `(src, band_index)` tuple
- A `numpy.ndarray`, 2D or 3D. If the array is 3D, ensure that it is in rasterio band order.

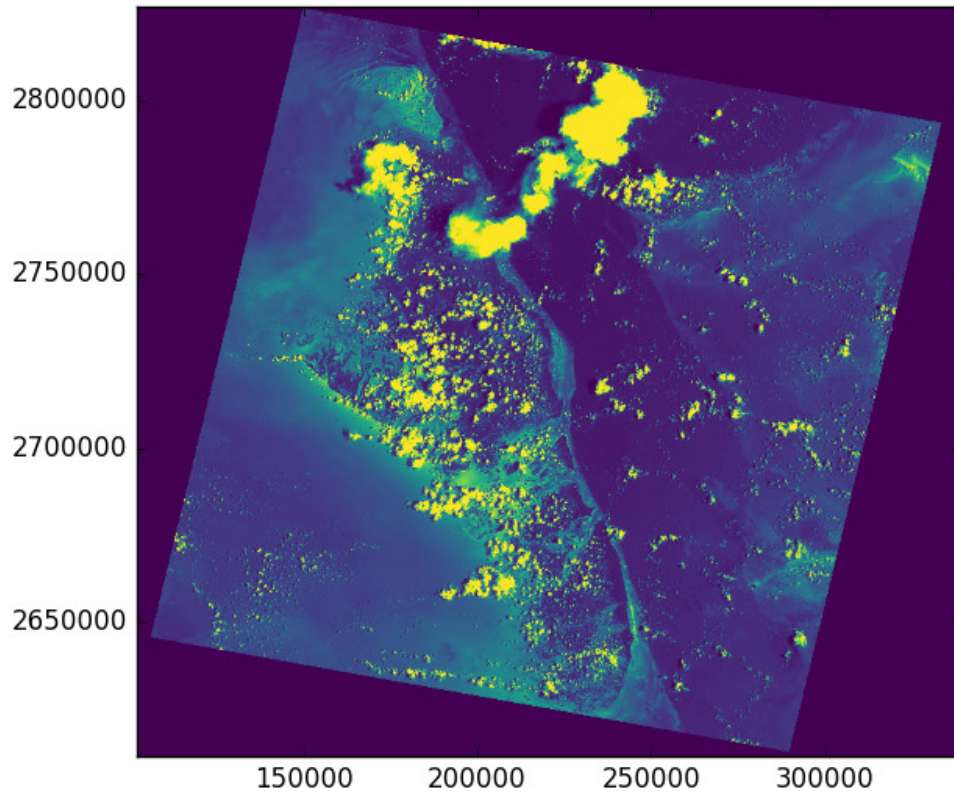
Thus the following operations for 3-band RGB data are equivalent. Note that when passing arrays, you can pass in a transform in order to get extent labels.

```
>>> from rasterio.plot import show
>>> show(src)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> show(src.read(), transform=src.transform)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
```



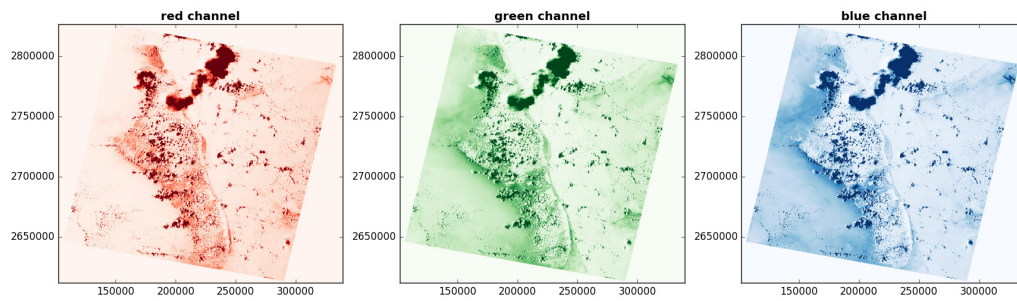
and similarly for single band plots. Note that you can pass in `cmap` to specify a matplotlib color ramp. Any kwargs passed to `show()` will be passed through to the underlying pyplot functions.

```
>>> show((src, 2), cmap='viridis')
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> show(src.read(2), transform=src.transform, cmap='viridis')
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
```



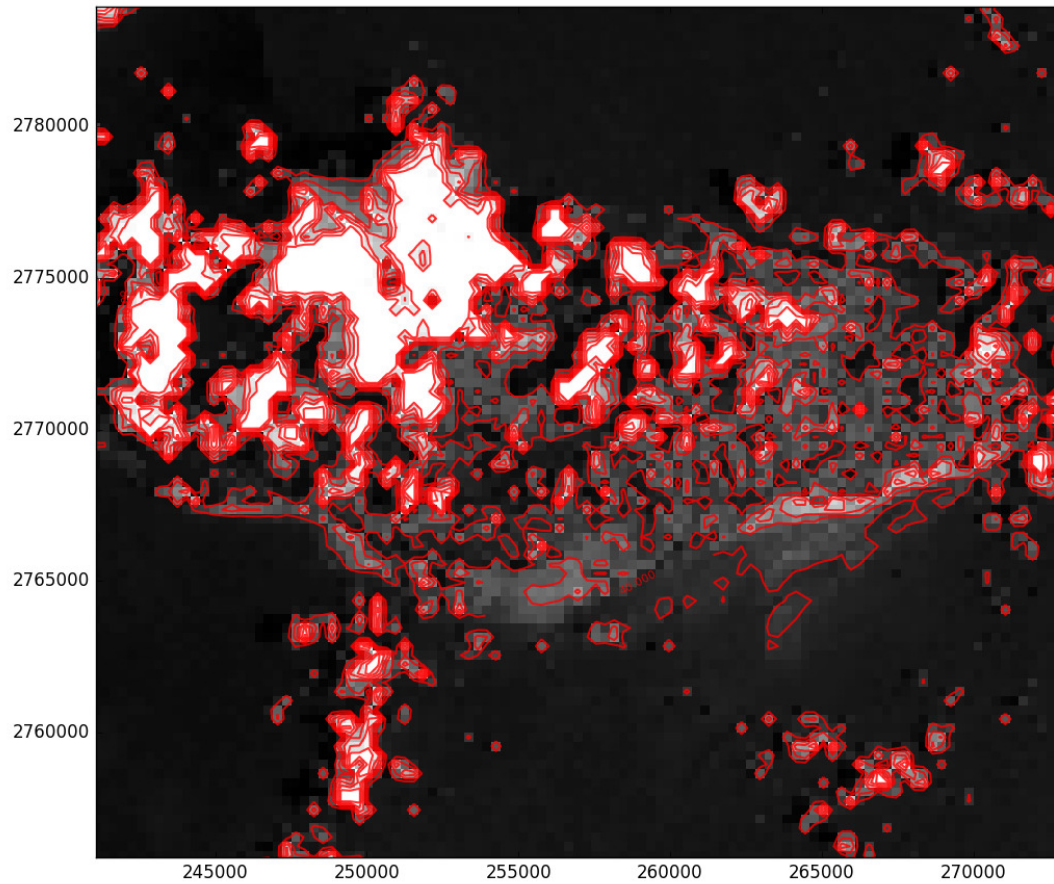
You can create a figure with multiple subplots by passing the `show(..., ax=ax1)` argument. Also note that this example demonstrates setting the overall figure size and sets a title for each subplot.

```
>>> fig, (axr, axg, axb) = pyplot.subplots(1,3, figsize=(21,7))
>>> show((src, 1), ax=axr, cmap='Reds', title='red channel')
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> show((src, 2), ax=axg, cmap='Greens', title='green channel')
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> show((src, 3), ax=axb, cmap='Blues', title='blue channel')
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> pyplot.show()
```



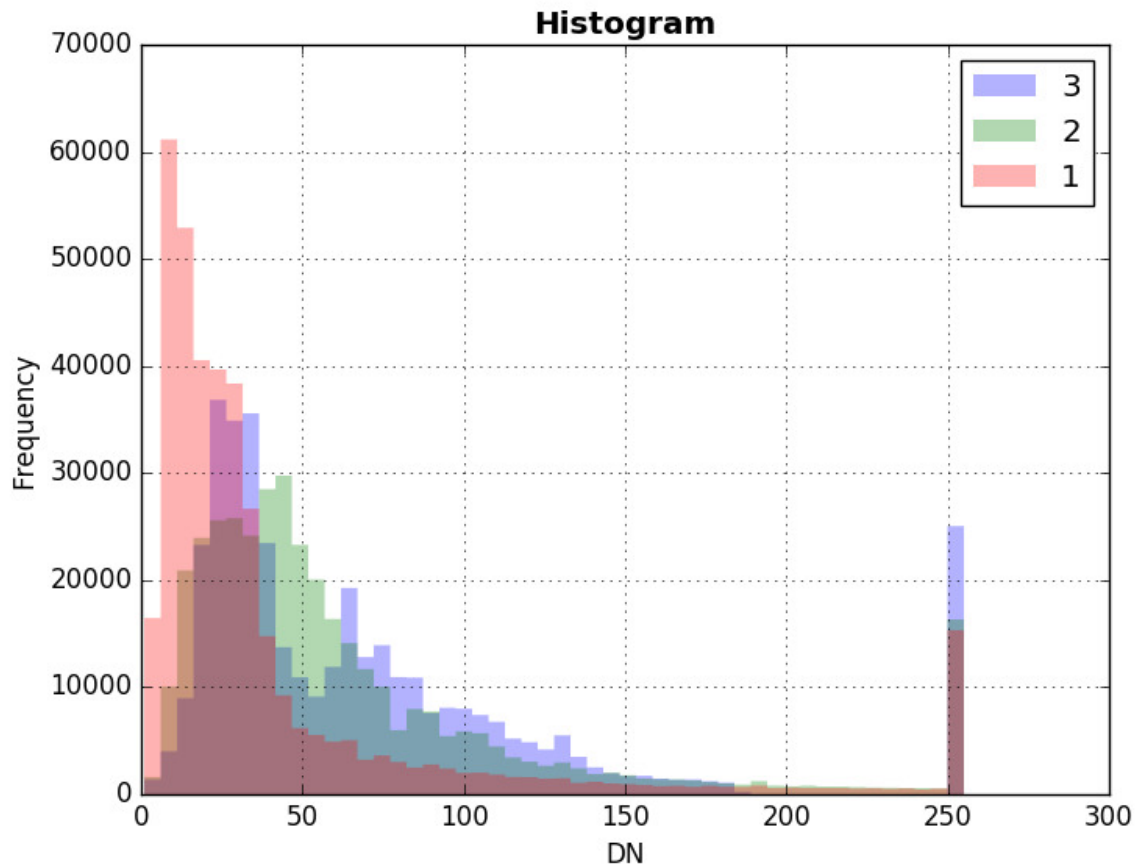
For single-band rasters, there is also an option to generate contours.

```
>>> fig, ax = pyplot.subplots(1, figsize=(12, 12))
>>> show((src, 1), cmap='Greys_r', interpolation='none', ax=ax)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> show((src, 1), contour=True, ax=ax)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> pyplot.show()
```



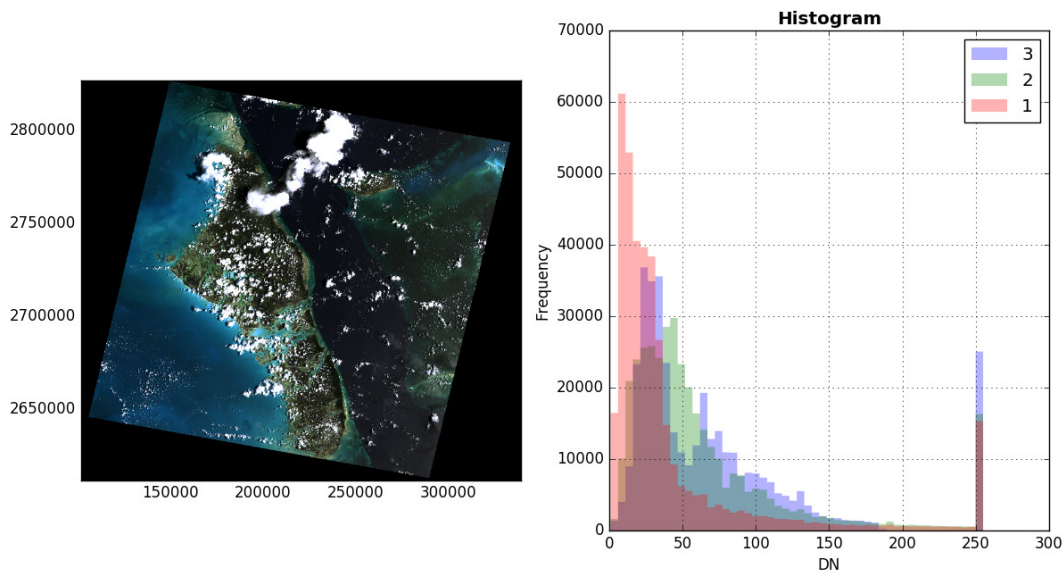
Rasterio also provides a `show_hist()` function for generating histograms of single or multiband rasters:

```
>>> from rasterio.plot import show_hist
>>> show_hist(
...     src, bins=50, lw=0.0, stacked=False, alpha=0.3,
...     histtype='stepfilled', title="Histogram")
```



The `show_hist()` function also takes an `ax` argument to allow subplot configurations

```
>>> fig, (axrgb, axhist) = pyplot.subplots(1, 2, figsize=(14,7))
>>> show(src, ax=axrgb)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>
>>> show_hist(src, bins=50, histtype='stepfilled',
...          lw=0.0, stacked=False, alpha=0.3, ax=axhist)
>>> pyplot.show()
```

5.18 Profiles and Writing Files

How to use profiles when opening files.

Like Python's built-in `open()` function, `rasterio.open()` has two primary arguments: a path (or URL) and an optional mode ('r', 'w', 'r+', or 'w+'). In addition there are a number of keyword arguments, several of which are required when creating a new dataset:

- driver
- width, height
- count
- dtype
- crs
- transform

These same parameters surface in a dataset's `profile` property. Exploiting the symmetry between a profile and dataset opening keyword arguments is good Rasterio usage.

```
with rasterio.open('first.jp2') as src_dataset:

    # Get a copy of the source dataset's profile. Thus our
    # destination dataset will have the same dimensions,
    # number of bands, data type, and georeferencing as the
    # source dataset.
    kwds = src_dataset.profile

    # Change the format driver for the destination dataset to
    # 'GTiff', short for GeoTIFF.
    kwds['driver'] = 'GTiff'
```

(continues on next page)

(continued from previous page)

```

# Add GeoTIFF-specific keyword arguments.
kwds['tiled'] = True
kwds['blockxsize'] = 256
kwds['blockysize'] = 256
kwds['photometric'] = 'YCbCr'
kwds['compress'] = 'JPEG'

with rasterio.open('second.tif', 'w', **kwds) as dst_dataset:
    # Write data to the destination dataset.

```

The `rasterio.profiles` module contains an example of a named profile that may be useful in applications:

```

class DefaultGtiffProfile(Profile):
    """Tiled, band-interleaved, LZW-compressed, 8-bit GTiff."""

    defaults = {
        'driver': 'GTiff',
        'interleave': 'band',
        'tiled': True,
        'blockxsize': 256,
        'blockysize': 256,
        'compress': 'lzw',
        'nodata': 0,
        'dtype': uint8
    }

```

It can be used to create new datasets. Note that it doesn't count bands and that a `count` keyword argument needs to be passed when creating a profile.

```

from rasterio.profiles import DefaultGtiffProfile

with rasterio.open(
    'output.tif', 'w', **DefaultGtiffProfile(count=3)) as dst_dataset:
    # Write data to the destination dataset.

```

5.19 Reading Datasets

Todo:

- **Discuss and/or link to topics**
 - supported formats, drivers
 - vsi
 - tags
 - profile
 - crs
 - transforms
 - dtypes

Dataset objects provide read, read-write, and write access to raster data files and are obtained by calling `rasterio.open()`. That function mimics Python's built-in `open()` and the dataset objects it returns mimic Python file objects.

```
>>> import rasterio
>>> src = rasterio.open('tests/data/RGB.byte.tif')
>>> src
<open DatasetReader name='tests/data/RGB.byte.tif' mode='r'>
>>> src.name
'tests/data/RGB.byte.tif'
>>> src.mode
'r'
>>> src.closed
False
```

If you try to access a nonexistent path, `rasterio.open()` does the same thing as `open()`, raising an exception immediately.

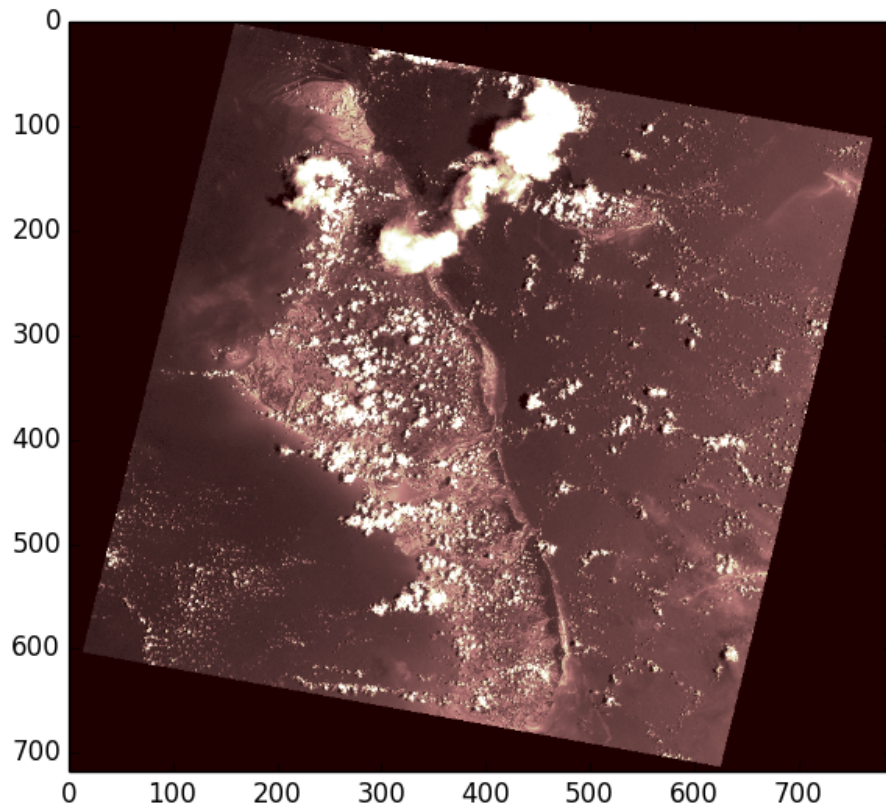
```
>>> open('/lol/wut.tif')
Traceback (most recent call last):
...
FileNotFoundError: [Errno 2] No such file or directory: '/lol/wut.tif'
>>> rasterio.open('/lol/wut.tif')
Traceback (most recent call last):
...
rasterio.errors.RasterioIOError: No such file or directory
```

Datasets generally have one or more bands (or layers). Following the GDAL convention, these are indexed starting with the number 1. The first band of a file can be read like this:

```
>>> array = src.read(1)
>>> array.shape
(718, 791)
```

The returned object is a 2-dimensional `numpy.ndarray`. The representation of that array at the Python prompt is a summary; the GeoTIFF file that Rasterio uses for testing has 0 values in the corners, but has nonzero values elsewhere.

```
>>> from matplotlib import pyplot
>>> pyplot.imshow(array, cmap='pink')
<matplotlib.image.AxesImage object at 0x...>
>>> pyplot.show()
```

Instead of reading single bands, all bands of the input dataset can be read into a 3-dimensional ndarray. Note that the interpretation of the 3 axes is (bands, rows, columns). See [Image processing software](#) for more details on how to convert to the ordering expected by some software.

```
>>> array = src.read()
>>> array.shape
(3, 718, 791)
```

In order to read smaller chunks of the dataset, refer to [Windowed reading and writing](#).

The indexes, Numpy data types, and nodata values of all a dataset's bands can be had from its `indexes`, `dtypes`, and `nodatavals` attributes.

```
>>> for i, dtype, nodataval in zip(src.indexes, src.dtypes, src.nodatavals):
...     print(i, dtype, nodataval)
...
1 uint8 0.0
2 uint8 0.0
3 uint8 0.0
```

To close a dataset, call its `close()` method.

```
>>> src.close()
>>> src
<closed DatasetReader name='tests/data/RGB.byte.tif' mode='r'>
```

After it's closed, data can no longer be read.

```
>>> src.read(1)
Traceback (most recent call last):
...
ValueError: can't read closed raster file
```

This is the same behavior as Python's file.

```
>>> f = open('README.rst')
>>> f.close()
>>> f.read()
Traceback (most recent call last):
...
ValueError: I/O operation on closed file.
```

As Python file objects can, Rasterio datasets can manage the entry into and exit from runtime contexts created using a `with` statement. This ensures that files are closed no matter what exceptions may be raised within the the block.

```
>>> with rasterio.open('tests/data/RGB.byte.tif', 'r') as one:
...     with rasterio.open('tests/data/RGB.byte.tif', 'r') as two:
...         print(two)
...         print(one)
...         raise Exception("an error occurred")
...
<open DatasetReader name='tests/data/RGB.byte.tif' mode='r'>
<open DatasetReader name='tests/data/RGB.byte.tif' mode='r'>
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: an error occurred
>>> print(two)
<closed DatasetReader name='tests/data/RGB.byte.tif' mode='r'>
>>> print(one)
<closed DatasetReader name='tests/data/RGB.byte.tif' mode='r'>
```

Format-specific dataset reading options may be passed as keyword arguments. For example, to turn off all types of GeoTIFF georeference except that within the TIFF file's keys and tags, pass `GEOREF_SOURCES='INTERNAL'`.

```
>>> with rasterio.open('tests/data/RGB.byte.tif', GEOREF_SOURCES='INTERNAL') as dataset:
...     # .aux.xml, .tab, .tfw sidecar files will be ignored.
```

5.20 Reprojection

Rasterio can map the pixels of a destination raster with an associated coordinate reference system and transform to the pixels of a source image with a different coordinate reference system and transform. This process is known as reprojection.

Rasterio's `rasterio.warp.reproject()` is a geospatial-specific analog to SciPy's `scipy.ndimage.interpolation.geometric_transform()`¹.

The code below reprojects between two arrays, using no pre-existing GIS datasets. `rasterio.warp.reproject()` has two positional arguments: source and destination. The remaining keyword arguments parameterize the reprojection

¹ https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.geometric_transform.html#scipy.ndimage.geometric_transform

transform.

```
import numpy as np
import rasterio
from rasterio import Affine as A
from rasterio.warp import reproject, Resampling

with rasterio.Env():

    # As source: a 512 x 512 raster centered on 0 degrees E and 0
    # degrees N, each pixel covering 15".
    rows, cols = src_shape = (512, 512)
    d = 1.0/240 # decimal degrees per pixel
    # The following is equivalent to
    # A(d, 0, -cols*d/2, 0, -d, rows*d/2).
    src_transform = A.translation(-cols*d/2, rows*d/2) * A.scale(d, -d)
    src_crs = {'init': 'EPSG:4326'}
    source = np.ones(src_shape, np.uint8)*255

    # Destination: a 1024 x 1024 dataset in Web Mercator (EPSG:3857)
    # with origin at 0.0, 0.0.
    dst_shape = (1024, 1024)
    dst_transform = A.translation(-237481.5, 237536.4) * A.scale(425.0, -425.0)
    dst_crs = {'init': 'EPSG:3857'}
    destination = np.zeros(dst_shape, np.uint8)

    reproject(
        source,
        destination,
        src_transform=src_transform,
        src_crs=src_crs,
        dst_transform=dst_transform,
        dst_crs=dst_crs,
        resampling=Resampling.nearest)

    # Assert that the destination is only partly filled.
    assert destination.any()
    assert not destination.all()
```

See `examples/reproject.py` for code that writes the destination array to a GeoTIFF file. I've uploaded the resulting file to a Mapbox map to show that the reprojection is correct: <https://a.tiles.mapbox.com/v3/sgillies.hfek2oko/page.html?secure=1#6/0.000/0.033>. (dead link)

5.20.1 Estimating optimal output shape

Rasterio provides a `rasterio.warp.calculate_default_transform()` function to determine the optimal resolution and transform for the destination raster. Given a source dataset in a known coordinate reference system, this function will return a transform, width, height tuple which is calculated by libgdal.

5.20.2 Reprojecting a GeoTIFF dataset

Reprojecting a GeoTIFF dataset from one coordinate reference system is a common use case. Rasterio provides a few utilities to make this even easier:

`transform_bounds()` transforms the bounding coordinates of the source raster to the target coordinate reference system, densifying points along the edges to account for non-linear transformations of the edges.

`calculate_default_transform()` transforms bounds to target coordinate system, calculates resolution if not provided, and returns destination transform and dimensions.

```
import numpy as np
import rasterio
from rasterio.warp import calculate_default_transform, reproject, Resampling

dst_crs = 'EPSG:4326'

with rasterio.open('rasterio/tests/data/RGB.byte.tif') as src:
    transform, width, height = calculate_default_transform(
        src.crs, dst_crs, src.width, src.height, *src.bounds)
    kwargs = src.meta.copy()
    kwargs.update({
        'crs': dst_crs,
        'transform': transform,
        'width': width,
        'height': height
    })

    with rasterio.open('/tmp/RGB.byte.wgs84.tif', 'w', **kwargs) as dst:
        for i in range(1, src.count + 1):
            reproject(
                source=rasterio.band(src, i),
                destination=rasterio.band(dst, i),
                src_transform=src.transform,
                src_crs=src.crs,
                dst_transform=transform,
                dst_crs=dst_crs,
                resampling=Resampling.nearest)
```

See `rasterio/rio/warp.py` for more complex examples of reprojection based on new bounds, dimensions, and resolution (as well as a command-line interface described [here](#)).

It is also possible to use `reproject()` to create an output dataset zoomed out by a factor of 2. Methods of the `rasterio.Affine` class help us generate the output dataset's transform matrix and, thereby, its spatial extent.

```
import numpy as np
import rasterio
from rasterio import Affine as A
```

(continues on next page)

(continued from previous page)

```

from rasterio.warp import reproject, Resampling

with rasterio.open('rasterio/tests/data/RGB.byte.tif') as src:
    src_transform = src.transform

    # Zoom out by a factor of 2 from the center of the source
    # dataset. The destination transform is the product of the
    # source transform, a translation down and to the right, and
    # a scaling.
    dst_transform = src_transform*A.translation(
        -src.width/2.0, -src.height/2.0)*A.scale(2.0)

    data = src.read()

    kwargs = src.meta
    kwargs['transform'] = dst_transform

    with rasterio.open('/tmp/zoomed-out.tif', 'w', **kwargs) as dst:

        for i, band in enumerate(data, 1):
            dest = np.zeros_like(band)

            reproject(
                band,
                dest,
                src_transform=src_transform,
                src_crs=src.crs,
                dst_transform=dst_transform,
                dst_crs=src.crs,
                resampling=Resampling.nearest)

            dst.write(dest, indexes=i)

```

5.20.3 Reprojecting with other georeferencing metadata

Most geospatial datasets have a geotransform which can be used to reproject a dataset from one coordinate reference system to another. Datasets may also be georeferenced by alternative metadata, namely Ground Control Points (gcps) or Rational Polynomial Coefficients (rpcs). For details on gcps and rpcs, see [Georeferencing](#). A common scenario is using gcps or rpcs to geocode (orthorectify) datasets, resampling and reorienting them to a coordinate reference system with a newly computed geotransform.

```

import numpy as np
import rasterio
from rasterio.warp import reproject
from rasterio.enums import Resampling

with rasterio.open('RGB.byte.rpc.vrt') as source:
    print(source.rpc)
    src_crs = "EPSG:4326" # This is the crs of the rpcs

```

(continues on next page)

(continued from previous page)

```

# Optional keyword arguments to be passed to GDAL transformer
# https://gdal.org/api/gdal_alg.html?highlight=gdalcreategenimgprojtransformer2#_
→CPPv432GDALCreateGenImgProjTransformer212GDALDatasetH12GDALDatasetHPPc
kwargs = {
    'RPC_DEM': '/path/to/dem.tif'
}

# Destination: a 1024 x 1024 dataset in Web Mercator (EPSG:3857)
destination = np.zeros((1024, 1024), dtype=np.uint8)
dst_crs = "EPSG:3857"

_, dst_transform = reproject(
    rasterio.band(source, 1),
    destination,
    rpcs=source.rpcs,
    src_crs=src_crs,
    dst_crs=dst_crs,
    resampling=Resampling.nearest,
    **kwargs
)

assert destination.any()

```

Note: When reprojecting a dataset with gcps or rpcs, the `src_crs` parameter should be supplied with the coordinate reference system that the gcps or rpcs are referenced against. By definition rpcs are always referenced against WGS84 ellipsoid with geographic coordinates (EPSG:4326)².

5.20.4 References

5.21 Resampling

For details on changing coordinate reference systems, see *Reprojection*.

5.21.1 Up and downsampling

Resampling refers to changing the cell values due to changes in the raster cell grid. This can occur during reprojection. Even if the projection is not changing, we may want to change the effective cell size of an existing dataset.

Upsampling refers to cases where we are converting to higher resolution/smaller cells. *Downsampling* is resampling to lower resolution/larger cellsizes.

By reading from a raster source into an output array of a different size or by specifying an `out_shape` of a different size you are effectively resampling the data.

Here is an example of upsampling by a factor of 2 using the bilinear resampling method.

² http://geotiff.maptools.org/rpc_prop.html

```

import rasterio
from rasterio.enums import Resampling

upscale_factor = 2

with rasterio.open("example.tif") as dataset:

    # resample data to target shape
    data = dataset.read(
        out_shape=(
            dataset.count,
            int(dataset.height * upscale_factor),
            int(dataset.width * upscale_factor)
        ),
        resampling=Resampling.bilinear
    )

    # scale image transform
    transform = dataset.transform * dataset.transform.scale(
        (dataset.width / data.shape[-1]),
        (dataset.height / data.shape[-2])
    )

```

Downsampling to 1/2 of the resolution can be done with `upscale_factor = 1/2`.

5.21.2 Resampling Methods

When you change the raster cell grid, you must recalculate the pixel values. There is no “correct” way to do this as all methods involve some interpolation.

The current resampling methods can be found in the `rasterio.enums.Resampling` class.

Of note, the default `nearest` method may not be suitable for continuous data. In those cases, `bilinear` and `cubic` are better suited. Some specialized statistical resampling method exist, e.g. `average`, which may be useful when certain numerical properties of the data are to be retained.

5.22 Switching from GDAL's Python bindings

This document is written specifically for users of GDAL's Python bindings (`osgeo.gdal`) who have read about Rasterio's *philosophy* and want to know what switching entails. The good news is that switching may not be complicated. This document explains the key similarities and differences between these two Python packages and highlights the features of Rasterio that can help in switching.

5.22.1 Mutual Incompatibilities

Rasterio and GDAL's bindings can contend for global GDAL objects. Unless you have deep knowledge about both packages, choose exactly one of `import osgeo.gdal` or `import rasterio`.

GDAL's bindings (`gdal` for the rest of this document) and Rasterio are not entirely compatible and should not, without a great deal of care, be imported and used in a single Python program. The reason is that the dynamic library they each load (these are C extension modules, remember), `libgdal.so` on Linux, `gdal.dll` on Windows, has a number of global objects and the two modules take different approaches to managing these objects.

Static linking of the GDAL library for `gdal` and `rasterio` can avoid this contention, but in practice you will almost never see distributions of these modules that statically link the GDAL library.

Beyond the issues above, the modules have different styles – `gdal` reads and writes like C while `rasterio` is more Pythonic – and don't complement each other well.

5.22.2 The GDAL Environment

GDAL library functions are executed in a context of format drivers, error handlers, and format-specific configuration options that this document will call the "GDAL Environment." Rasterio has an abstraction for the GDAL environment, `gdal` does not.

With `gdal`, this context is initialized upon import of the module. This makes sense because `gdal` objects are thin wrappers around functions and classes in the GDAL dynamic library that generally require registration of drivers and error handlers. The `gdal` module doesn't have an abstraction for the environment, but it can be modified using functions like `gdal.SetErrorHandler()` and `gdal.UseExceptions()`.

Rasterio has modules that don't require complete initialization and configuration of GDAL (`rasterio.dtypes`, `rasterio.profiles`, and `rasterio.windows`, for example) and in the interest of reducing overhead doesn't register format drivers and error handlers until they are needed. The functions that do need fully initialized GDAL environments will ensure that they exist. `rasterio.open()` is the foremost of this category of functions. Consider the example code below.

```
import rasterio
# The GDAL environment has no registered format drivers or error
# handlers at this point.

with rasterio.open('example.tif') as src:
    # Format drivers and error handlers are registered just before
    # open() executes.
```

Importing `rasterio` does not initialize the GDAL environment. Calling `rasterio.open()` does. This is different from `gdal` where `import osgeo.gdal`, not `osgeo.gdal.Open()`, initializes the GDAL environment.

Rasterio has an abstraction for the GDAL environment, `rasterio.Env`, that can be invoked explicitly for more control over the configuration of GDAL as shown below.


```

import rasterio
# The GDAL environment has no registered format drivers or error
# handlers at this point.

with rasterio.Env(CPL_DEBUG=True, GDAL_CACHEMAX=128000000):
    # This ensures that all drivers are registered in the global
    # context. Within this block *only* GDAL's debugging messages
    # are turned on and the raster block cache size is set to 128 MB.

    with rasterio.open('example.tif') as src:
        # Perform GDAL operations in this context.
        # ...
        # Done.

# At this point, configuration options are set back to their
# previous (possibly unset) values. The raster block cache size
# is returned to its default (5% of available RAM) and debugging
# messages are disabled.

```

As mentioned previously, `gdal` has no such abstraction for the GDAL environment. The nearest approximation would be something like the code below.

```

from osgeo import gdal

# Define a new configuration, save the previous configuration,
# and then apply the new one.
new_config = {
    'CPL_DEBUG': 'ON', 'GDAL_CACHEMAX': '512'}
prev_config = {
    key: gdal.GetConfigOption(key) for key in new_config.keys()}
for key, val in new_config.items():
    gdal.SetConfigOption(key, val)

# Perform GDAL operations in this context.
# ...
# Done.

# Restore previous configuration.
for key, val in prev_config.items():
    gdal.SetConfigOption(key, val)

```

Rasterio achieves this with a single Python statement.

```

with rasterio.Env(CPL_DEBUG=True, GDAL_CACHEMAX=512000000):
    # ...

```

Please note that to the `Env` class, `GDAL_CACHEMAX` is strictly an integer number of bytes. GDAL's shorthand notation is not supported.

5.22.3 Format Drivers

`gdal` provides objects for each of the GDAL format drivers. With Rasterio, format drivers are represented by strings and are used only as arguments to functions like `rasterio.open()`.

```
dst = rasterio.open('new.tif', 'w', format='GTiff', **kwargs)
```

Rasterio uses the same format driver names as GDAL does.

5.22.4 Dataset Identifiers

Rasterio uses URIs to identify datasets, with schemes for different protocols. The GDAL bindings have their own special syntax.

Unix-style filenames such as `/var/data/example.tif` identify dataset files for both Rasterio and `gdal`. Rasterio also accepts ‘file’ scheme URIs like `file:///var/data/example.tif`.

Rasterio identifies datasets within ZIP or tar archives using Apache VFS style identifiers like `zip:///var/data/example.zip!example.tif` or `tar:///var/data/example.tar!example.tif`.

Datasets served via HTTPS are identified using ‘https’ URIs like `https://landsat-pds.s3.amazonaws.com/L8/139/045/LC81390452014295LGN00/LC81390452014295LGN00_B1.TIF`.

Datasets on AWS S3 are identified using ‘s3’ scheme identifiers like `s3://landsat-pds/L8/139/045/LC81390452014295LGN00/LC81390452014295LGN00_B1.TIF`.

With `gdal`, the equivalent identifiers are respectively `/vsizip//var/data/example.zip/example.tif`, `/vsitar//var/data/example.tar/example.tif`, `/vsicurl/landsat-pds.s3.amazonaws.com/L8/139/045/LC81390452014295LGN00/LC81390452014295LGN00_B1.TIF`, and `/vsis3/landsat-pds/L8/139/045/LC81390452014295LGN00/LC81390452014295LGN00_B1.TIF`.

To help developers switch, Rasterio will accept these identifiers and other format-specific connection strings, too, and dispatch them to the proper format drivers and protocols.

5.22.5 Dataset Objects

Rasterio and `gdal` each have dataset objects. Not the same classes, of course, but not radically different ones. In each case, you generally get dataset objects through an “opener” function: `rasterio.open()` or `gdal.Open()`.

So that Python developers can spend less time reading docs, the dataset object returned by `rasterio.open()` is modeled on Python’s file object. It even has the `close()` method that `gdal` lacks so that you can actively close dataset connections.

5.22.6 Bands

`gdal` and Rasterio both have band objects. But unlike `gdal`’s band, Rasterio’s band is just a tuple of the dataset, band index and some other band properties. Thus Rasterio never has objects with dangling dataset pointers. With Rasterio, bands are represented by a numerical index, starting from 1 (as GDAL does), and are used as arguments to dataset methods. To read the first band of a dataset as a `numpy.ndarray`, do this.

```
with rasterio.open('example.tif') as src:
    band1 = src.read(1)
```

A band object can be used to represent a single band (or a sequence of bands):

```
with rasterio.open('example.tif') as src:
    bnd = rasterio.band(src, 1)
    print(bnd.dtype)
```

Other attributes of GDAL band objects generally surface in Rasterio as tuples returned by dataset attributes, with one value per band, in order.

```
>>> src = rasterio.open('example.tif')
>>> src.indexes
(1, 2, 3)
>>> src.dtypes
('uint8', 'uint8', 'uint8')
>>> src.descriptions
('Red band', 'Green band', 'Blue band')
>>> src.units
('DN', 'DN', 'DN')
```

Developers that want read-only band objects for their applications can create them by zipping these tuples together.

```
from collections import namedtuple

Band = namedtuple('Band', ['idx', 'dtype', 'description', 'units'])

src = rasterio.open('example.tif')
bands = [Band(vals) for vals in zip(
    src.indexes, src.dtypes, src.descriptions, src.units)]
```

Namedtuples are like lightweight classes.

```
>>> for band in bands:
...     print(band.idx)
...
1
2
3
```

5.22.7 Geotransforms

The `DatasetReader.transform` attribute is comparable to the `GeoTransform` attribute of a GDAL dataset, but Rasterio's has more power. It's not just an array of affine transformation matrix elements, it's an instance of an `Affine` class and has many handy methods. For example, the spatial coordinates of the upper left corner of any raster element is the product of the `DatasetReader.transform` matrix and the (column, row) index of the element.

```
>>> src = rasterio.open('example.tif')
>>> src.transform * (0, 0)
(101985.0, 2826915.0)
```

The affine transformation matrix can be inverted as well.

```
>>> ~src.transform * (101985.0, 2826915.0)
(0.0, 0.0)
```

To help developers switch, `Affine` instances can be created from or converted to the sequences used by `gdal`.

```
>>> from rasterio.transform import Affine
>>> Affine.from_gdal(101985.0, 300.0379266750948, 0.0,
...                 2826915.0, 0.0, -300.041782729805).to_gdal()
...
(101985.0, 300.0379266750948, 0.0, 2826915.0, 0.0, -300.041782729805)
```

5.22.8 Coordinate Reference Systems

The `DatasetReader.crs` attribute is an instance of Rasterio's `CRS()` class and works well with `pyproj`.

```
>>> from pyproj import Transformer
>>> src = rasterio.open('example.tif')
>>> transformer = Transformer.from_crs(src.crs, "EPSG:3857", always_xy=True)
>>> transformer.transform(101985.0, 2826915.0)
(-8789636.707871985, 2938035.238323653)
```

5.22.9 Tags

GDAL metadata items are called “tags” in Rasterio. The tag set for a given GDAL metadata namespace is represented as a dict.

```
>>> src.tags()
{'AREA_OR_POINT': 'Area'}
>>> src.tags(ns='IMAGE_STRUCTURE')
{'INTERLEAVE': 'PIXEL'}
```

The semantics of the tags in GDAL's default and `IMAGE_STRUCTURE` namespaces are described in https://gdal.org/user/raster_data_model.html. Rasterio uses several namespaces of its own: `rio_creation_kwds` and `rio_overviews`, each with their own semantics.

5.22.10 Offsets and Windows

Rasterio adds an abstraction for subsets or windows of a raster array that GDAL does not have. A window is a pair of tuples, the first of the pair being the raster row indexes at which the window starts and stops, the second being the column indexes at which the window starts and stops. Row before column, as with `ndarray` slices. Instances of `Window` are created by passing the four subset parameters used with `gdal` to the class constructor.

```
src = rasterio.open('example.tif')

xoff, yoff = 0, 0
xsize, ysize = 10, 10
subset = src.read(1, window=Window(xoff, yoff, xsize, ysize))
```

5.22.11 Valid Data Masks

Rasterio provides an array for every dataset representing its valid data mask using the same indicators as GDAL: 0 for invalid data and 255 for valid data.

```
>>> src = rasterio.open('example.tif')
>>> src.dataset_mask()
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

Arrays for dataset bands can also be had as a `numpy.ma.MaskedArray`.

```
>>> src.read(1, masked=True)
masked_array(data =
  [[-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]
  ...,
  [-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]
  [-- -- -- ..., -- -- --]],
            mask =
  [[ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]
  ...,
  [ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]
  [ True  True  True ...,  True  True  True]],
            fill_value = 0)
```

Where the masked array's mask is True, the data is invalid and has been masked “out” in the opposite sense of GDAL's mask.

5.22.12 Errors and Exceptions

Rasterio always raises Python exceptions when an error occurs and never returns an error code or `None` to indicate an error. `gdal` takes the opposite approach, although developers can turn on exceptions by calling `gdal.UseExceptions()`.

5.23 Tagging datasets and bands

GDAL's [data model](#) includes collections of key, value pairs for major classes. In that model, these are “metadata”, but since they don't have to be just for metadata, these key, value pairs are called “tags” in rasterio.

5.23.1 Reading tags

I'm going to use the rasterio interactive inspector in these examples below.

```
$ rio insp tests/data/RGB.byte.tif
Rasterio 1.2.0 Interactive Inspector (Python 3.7.8)
Type "src.name", "src.read(1)", or "help(src)" for more information.
>>>
```

Tags belong to namespaces. To get a copy of a dataset's tags from the default namespace, call `tags()` with no arguments.

```
>>> import rasterio
>>> src = rasterio.open("tests/data/RGB.byte.tif")
>>> src.tags()
{'AREA_OR_POINT': 'Area'}
```

A dataset's bands may have tags, too. Here are the tags from the default namespace for the first band, accessed using the positional band index argument of `tags()`.

```
>>> src.tags(1)['STATISTICS_MEAN']
'29.947726688477'
```

These are the tags that came with the sample data I'm using to test rasterio. In practice, maintaining stats in the tags can be unreliable as there is no automatic update of the tags when the band's image data changes.

The 3 standard, non-default GDAL tag namespaces are 'SUBDATASETS', 'IMAGE_STRUCTURE', and 'RPC'. You can get the tags from these namespaces using the `ns` keyword of `tags()`.

```
>>> src.tags(ns='IMAGE_STRUCTURE')
{'INTERLEAVE': 'PIXEL'}
>>> src.tags(ns='SUBDATASETS')
{}
>>> src.tags(ns='RPC')
{}
```

A special case for GDAL tag namespaces are those prefixed with 'xml' e.g. 'xml:TRE' or 'xml:VRT'. GDAL will treat these namespaces as a single xml string.

5.23.2 Writing tags

You can add new tags to a dataset or band, in the default or another namespace, using the `update_tags()` method. Unicode tag values, too, at least for TIFF files.

```
import rasterio

with rasterio.open(
    '/tmp/test.tif',
    'w',
    driver='GTiff',
    count=1,
    dtype=rasterio.uint8,
    width=10,
    height=10) as dst:

    dst.update_tags(a='1', b='2')
    dst.update_tags(1, c=3)
    with pytest.raises(ValueError):
        dst.update_tags(4, d=4)

    # True
    assert dst.tags() == {'a': '1', 'b': '2'}
    # True
    assert dst.tags(1) == {'c': '3' }

    dst.update_tags(ns='rasterio_testing', rus=u' ')
    # True
    assert dst.tags(ns='rasterio_testing') == {'rus': u' '}
```

As with image data, tags aren't written to the file on disk until the dataset is closed.

5.24 Transforms

Rasterio supports three primary methods for transforming of coordinates from image pixel (row, col) to and from geographic/projected (x, y) coordinates. The interface for performing these coordinate transformations is available in `rasterio.transform` through one of `AffineTransformer`, `GCPTransformer`, or `RPCTransformer`. The methods `xy()` and `rowcol()` are responsible for converting between (row, col) -> (x, y) and (x, y) -> (row, col), respectively.

5.24.1 Using Affine transformation matrix

`AffineTransformer` takes care of coordinate transformations given an Affine transformation matrix. For example

```
>>> transform = Affine(300.0379266750948, 0.0, 101985.0, 0.0,
                       -300.041782729805, 2826915.0)
>>> transformer = rasterio.transform.AffineTransformer(transform)
>>> transformer.xy(0, 0)
(102135.01896333754, 2826764.979108635)
>>> transformer.rowcol(102135.01896333754, 2826764.979108635)
(0, 0)
```

This is approximately equivalent to

```
>>> transform = Affine(300.0379266750948, 0.0, 101985.0, 0.0,
                       -300.041782729805, 2826915.0)
>>> transform * (0.5, 0.5)
(102135.01896333754, 2826764.979108635)
>>> ~transform * (102135.01896333754, 2826764.979108635)
(0.5, 0.5)
```

The dataset methods `xy()` and `index()` use `rasterio.transform` under the hood

```
>>> with rasterio.open('RGB.byte.tif') as src:
    print(src.xy(0, 0))
(102135.01896333754, 2826764.979108635)
```

5.24.2 Using Ground Control Points

```
>>> gcps = [GroundControlPoint(row=11521.5, col=0.5, x=-123.6185142817931, y=48.
    ↪ 99561141948625, z=89.13533782958984, id='217', info=''),
            GroundControlPoint(row=11521.5, col=7448.5, x=-122.8802747777599, y=48.
    ↪ 91210259315549, z=89.13533782958984, id='234', info=''),
            GroundControlPoint(row=0.5, col=0.5, x=-123.4809665720148, y=49.
    ↪ 52809729106944, z=89.13533782958984, id='1', info=''),
            GroundControlPoint(row=0.5, col=7448.5, x=-122.7345733674704, y=49.
    ↪ 44455878004666, z=89.13533782958984, id='18', info='')]
>>> transformer = rasterio.transform.GCPTransformer(gcps)
>>> transformer.xy(0, 0)
(-123.478928146887, 49.52808986989645)
```

5.24.3 Using Rational Polynomial Coefficients

For accuracy a height value is typically required when using `RPCTransformer`. By default, a value of 0 is assumed.

```
>>> with rasterio.open('RGB.byte.rpc.vrt') as src:
    transformer = rasterio.transform.RPCTransformer(src.rpcs)
    transformer.xy(0, 0)
(-123.47959047080701, 49.52794990575094)
```

A first order correction would be to use a mean elevation value for the image

```
>>> with rasterio.open('RGB.byte.rpc.vrt') as src:
    transformer = rasterio.transform.RPCTransformer(src.rpcs)
    transformer.xy(0, 0, zs=src.rpcs.height_off)
(-123.48096552376548, 49.528097381526386)
```

Better yet is to sample height values from a digital elevation model (DEM). `RPCTransformer` allows for options to be passed to `GDALCreateRPCTransformerV2()`

```
>>> with rasterio.open('RGB.byte.rpc.vrt') as src:
    transformer = rasterio.transform.RPCTransformer(src.rpcs, rpc_dem='vancouver-dem.
    ↪ tif')
    transformer.xy(0, 0)
(-123.47954729595642, 49.5279448909449)
```


5.24.4 Transformer Resources

The `AffineTransformer` is a pure Python class, however `GCPTransformer` and `RPCTransformer` make use of C/C++ GDAL objects. Explicit control of the transformer object can be achieved by use within a context manager or by calling `close()` method e.g.

```
>>> with rasterio.transform.RPCTransformer(rpcs) as transform:
    transform.xy(0, 0)
>>> transform.xy(0, 0)
ValueError: Unexpected NULL transformer
```

Note: If `RPC_DEM` is specified in `rpc_options`, GDAL will maintain an open file handle to the DEM until the transformer is closed.

5.25 Virtual Warping

Rasterio has a `WarpedVRT` class that abstracts many of the details of raster warping by using an in-memory `WarpedVRT`. A `WarpedVRT` can be the easiest solution for tiling large datasets.

For example, to virtually warp the `RGB.byte.tif` test dataset from its proper EPSG:32618 coordinate reference system to EPSG:3857 (Web Mercator) and extract pixels corresponding to its central zoom 9 tile, do the following.

```
from affine import Affine
import mercantile

import rasterio
from rasterio.enums import Resampling
from rasterio.vrt import WarpedVRT

with rasterio.open('tests/data/RGB.byte.tif') as src:
    with WarpedVRT(src, crs='EPSG:3857',
                    resampling=Resampling.bilinear) as vrt:

        # Determine the destination tile and its mercator bounds using
        # functions from the mercantile module.
        dst_tile = mercantile.tile(*vrt.lnglat(), 9)
        left, bottom, right, top = mercantile.xy_bounds(*dst_tile)

        # Determine the window to use in reading from the dataset.
        dst_window = vrt.window(left, bottom, right, top)

        # Read into a 3 x 512 x 512 array. Our output tile will be
        # 512 wide x 512 tall.
        data = vrt.read(window=dst_window, out_shape=(3, 512, 512))

        # Use the source's profile as a template for our output file.
        profile = vrt.profile
        profile['width'] = 512
        profile['height'] = 512
        profile['driver'] = 'GTiff'
```

(continues on next page)

(continued from previous page)

```

# We need determine the appropriate affine transformation matrix
# for the dataset read window and then scale it by the dimensions
# of the output array.
dst_transform = vrt.window_transform(dst_window)
scaling = Affine.scale(dst_window.height / 512,
                       dst_window.width / 512)
dst_transform *= scaling
profile['transform'] = dst_transform

# Write the image tile to disk.
with rasterio.open('/tmp/test-tile.tif', 'w', **profile) as dst:
    dst.write(data)

```

5.25.1 Normalizing Data to a Consistent Grid

A WarpedVRT can be used to normalize a stack of images with differing projections, bounds, cell sizes, or dimensions against a regular grid in a defined bounding box.

The `tests/data/RGB.byte.tif` file is in UTM zone 18, so another file in a different CRS is required for demonstration. This command will create a new image with drastically different dimensions and cell size, and reproject to WGS84. As of this writing `rio warp` implements only a subset of `gdalwarp`'s features, so `gdalwarp` must be used to achieve the desired transform:

```

$ gdalwarp \
  -t_srs EPSG:4326 \
  -te_srs EPSG:32618 \
  -te 101985 2673031 339315 2801254 \
  -ts 200 250 \
  tests/data/RGB.byte.tif \
  tests/data/WGS84-RGB.byte.tif

```

So, the attributes of these two images drastically differ:

```

$ rio info --shape tests/data/RGB.byte.tif
718 791
$ rio info --shape tests/data/WGS84-RGB.byte.tif
250 200
$ rio info --crs tests/data/RGB.byte.tif
EPSG:32618
$ rio info --crs tests/data/WGS84-RGB.byte.tif
EPSG:4326
$ rio bounds --bbox --geographic --precision 7 tests/data/RGB.byte.tif
[-78.95865, 23.5649912, -76.5749237, 25.5508738]
$ rio bounds --bbox --geographic --precision 7 tests/data/WGS84-RGB.byte.tif
[-78.9147773, 24.119606, -76.5963819, 25.3192311]

```

and this snippet demonstrates how to normalize data to consistent dimensions, CRS, and cell size within a pre-defined bounding box:

```

from __future__ import division

```

(continues on next page)

(continued from previous page)

```

import os

import affine

import rasterio
from rasterio.crs import CRS
from rasterio.enums import Resampling
from rasterio import shutil as rio_shutil
from rasterio.vrt import WarpedVRT

input_files = (
    # This file is in EPSG:32618
    'tests/data/RGB.byte.tif',
    # This file is in EPSG:4326
    'tests/data/WGS84-RGB.byte.tif'
)

# Destination CRS is Web Mercator
dst_crs = CRS.from_epsg(3857)

# These coordinates are in Web Mercator
dst_bounds = -8744355, 2768114, -8559167, 2908677

# Output image dimensions
dst_height = dst_width = 100

# Output image transform
left, bottom, right, top = dst_bounds
xres = (right - left) / dst_width
yres = (top - bottom) / dst_height
dst_transform = affine.Affine(xres, 0.0, left,
                              0.0, -yres, top)

vrt_options = {
    'resampling': Resampling.cubic,
    'crs': dst_crs,
    'transform': dst_transform,
    'height': dst_height,
    'width': dst_width,
}

for path in input_files:
    with rasterio.open(path) as src:

        with WarpedVRT(src, **vrt_options) as vrt:

            # At this point 'vrt' is a full dataset with dimensions,
            # CRS, and spatial extent matching 'vrt_options'.

            # Read all data into memory.

```

(continues on next page)

(continued from previous page)

```

data = vrt.read()

# Process the dataset in chunks. Likely not very efficient.
for _, window in vrt.block_windows():
    data = vrt.read(window=window)

# Dump the aligned data into a new file. A VRT representing
# this transformation can also be produced by switching
# to the VRT driver.
directory, name = os.path.split(path)
outfile = os.path.join(directory, 'aligned-{}'.format(name))
rio_shutil.copy(vrt, outfile, driver='GTiff')

```

5.26 Virtual Filesystems

Todo: Support for URIs describing zip, s3, https resources. Relationship to GDAL vsicurl, vsis3 et al.

Rasterio relies on GDAL's virtual filesystem interface to access datasets on the web, in cloud storage, in archive files, and in Python objects.

5.26.1 AWS S3

This is an extra feature that must be installed by executing

```
pip install rasterio[s3]
```

After you have configured your AWS credentials as explained in the [boto3 guide](#) you can read metadata and imagery from TIFFs stored as S3 objects with no change to your code.

```

with rasterio.open('s3://landsat-pds/L8/139/045/LC81390452014295LGN00/
↳LC81390452014295LGN00_B1.TIF') as src:
    print(src.profile)

# Printed:
# {'blockxsize': 512,
#  'blockysize': 512,
#  'compress': 'deflate',
#  'count': 1,
#  'crs': {'init': u'epsg:32645'},
#  'driver': u'GTiff',
#  'dtype': 'uint16',
#  'height': 7791,
#  'interleave': 'band',
#  'nodata': None,
#  'tiled': True,
#  'transform': Affine(30.0, 0.0, 381885.0,
#  0.0, -30.0, 2512815.0),
#  'width': 7621}

```

Note: AWS pricing concerns While this feature can reduce latency by reading fewer bytes from S3 compared to downloading the entire TIFF and opening locally, it does make at least 3 GET requests to fetch a TIFF’s *profile* as shown above and likely many more to fetch all the imagery from the TIFF. Consult the AWS S3 pricing guidelines before deciding if `aws.Session` is for you.

5.26.2 Python file openers

Datasets stored in proprietary systems or addressable only through protocols not directly supported by GDAL can be accessed using the `opener` keyword argument of `rasterio.open`. Here is an example of using `fs_s3fs` to access the dataset in `sentinel-s2-l2a-cogs/45/C/VQ/2022/11/S2B_45CVQ_20221102_0_L2A/B01.tif` from the `sentinel-cogs` AWS S3 bucket. Rasterio can access this without using the `opener` argument, but it makes a good usage example. Other custom openers would work in the same way.

```
import rasterio
from fs_s3fs import S3FS

fs = S3FS(
    bucket_name="sentinel-cogs",
    dir_path="sentinel-s2-l2a-cogs/45/C/VQ/2022/11/S2B_45CVQ_20221102_0_L2A",
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
)

with rasterio.open("B01.tif", opener=fs.openbin) as src:
    print(src.profile)
```

Where `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are placeholders for the appropriate credentials.

5.27 Windowed reading and writing

Beginning in rasterio 0.3, you can read and write “windows” of raster files. This feature allows you to work on rasters that are larger than your computers RAM or process chunks of large rasters in parallel.

5.27.1 Windows

A Window is a view onto a rectangular subset of a raster dataset and is described in rasterio by column and row offsets and width and height in pixels. These may be ints or floats.

```
from rasterio.windows import Window

Window(col_off, row_off, width, height)
```

Windows may also be constructed from numpy array index tuples or slice objects. Only int values are permitted in these cases.

```
Window.from_slices((row_start, row_stop), (col_start, col_stop))
Window.from_slices(slice(row_start, row_stop), slice(col_start, col_stop))
```

If height and width keyword arguments are passed to `from_slices()`, relative and open-ended slices may be used.

```
Window.from_slices(slice(None), slice(None), height=100, width=100)
# Window(col_off=0.0, row_off=0.0, width=100.0, height=100.0)

Window.from_slices(slice(10, -10), slice(10, -10), height=100, width=100)
# Window(col_off=10, row_off=10, width=80, height=80)
```

5.27.2 Reading

Here is an example of reading a 256 row x 512 column subset of the rasterio test file.

```
>>> import rasterio
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     w = src.read(1, window=Window(0, 0, 512, 256))
...
>>> print(w.shape)
(256, 512)
```

Attention: In getting data to fill a window Rasterio will read the entirety of one or more chunks of data from the dataset. If you're reading from a GeoTIFF with 512 x 512 pixel chunks (blocks), that determines the minimum number of bytes that will be read from disk or copied over your network, even if your read window is only 1 x 1 pixels. In the case that your source dataset does not use chunks (rare, but possible) Rasterio will read the entire dataset in order to fill even a 1 x 1 pixel window. In practice, it's important to chunk the data you create and store for your applications.

5.27.3 Writing

Writing works similarly. The following creates a blank 500 column x 300 row GeoTIFF and plops 37,500 pixels with value 127 into a window 30 pixels down from and 50 pixels to the right of the upper left corner of the GeoTIFF.

```
image = numpy.ones((150, 250), dtype=rasterio.ubyte) * 127

with rasterio.open(
    '/tmp/example.tif', 'w',
    driver='GTiff', width=500, height=300, count=1,
    dtype=image.dtype) as dst:
    dst.write(image, window=Window(50, 30, 250, 150), indexes=1)
```

The result:



5.27.4 Decimation

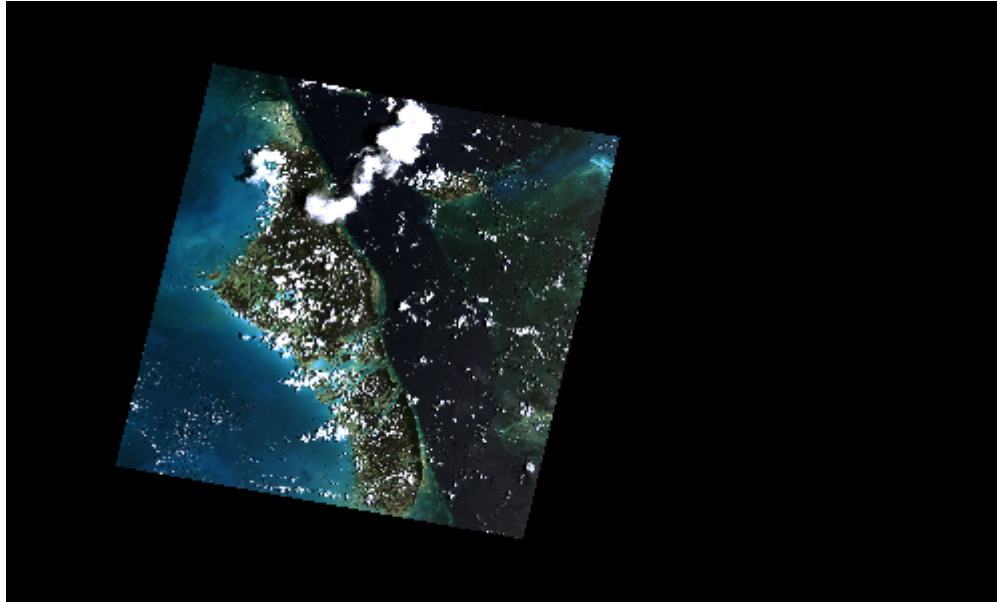
If the write window is smaller than the data, the data will be decimated. Below, the window is scaled to one third of the source image.

```
with rasterio.open('tests/data/RGB.byte.tif') as src:
    b, g, r = (src.read(k) for k in (1, 2, 3))
    # src.height = 718, src.width = 791

    write_window = Window.from_slices((30, 269), (50, 313))
    # write_window.height = 239, write_window.width = 263

    with rasterio.open(
        '/tmp/example.tif', 'w',
        driver='GTiff', width=500, height=300, count=3,
        dtype=r.dtype) as dst:
        for k, arr in [(1, b), (2, g), (3, r)]:
            dst.write(arr, indexes=k, window=write_window)
```

And the result:



5.27.5 Data windows

Sometimes it is desirable to crop off an outer boundary of NODATA values around a dataset. You can do this with `get_data_window()`:

```
from rasterio.windows import get_data_window

with rasterio.open('tests/data/RGB.byte.tif') as src:
    window = get_data_window(src.read(1, masked=True))
    # window = Window(col_off=13, row_off=3, width=757, height=711)

    kwargs = src.meta.copy()
    kwargs.update({
        'height': window.height,
        'width': window.width,
        'transform': rasterio.windows.transform(window, src.transform)})

    with rasterio.open('/tmp/cropped.tif', 'w', **kwargs) as dst:
        dst.write(src.read(window=window))
```

5.27.6 Window transforms

The affine transform of a window can be accessed using a dataset's `window_transform()` method:

```
>>> import rasterio
>>> from rasterio.windows import Window
>>> win = Window(256, 256, 128, 128)
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     src_transform = src.transform
...     win_transform = src.window_transform(win)
... 
```

(continues on next page)

(continued from previous page)

```
>>> print(src_transform)
| 300.04, 0.00, 101985.00|
| 0.00,-300.04, 2826915.00|
| 0.00, 0.00, 1.00|
>>> print(win_transform)
| 300.04, 0.00, 178794.71|
| 0.00,-300.04, 2750104.30|
| 0.00, 0.00, 1.00|
```

5.27.7 Window utilities

Basic union and intersection operations are available for windows, to streamline operations across dynamically created windows for a series of bands or datasets with the same full extent.

```
>>> from rasterio import windows
>>> # Full window is ((0, 1000), (0, 500))
>>> window1 = Window(10, 100, 490, 400)
>>> window2 = Window(50, 10, 200, 140)
>>> windows.union(window1, window2)
Window(col_off=10, row_off=10, width=490, height=490)
>>> windows.intersection(window1, window2)
Window(col_off=50, row_off=100, width=200, height=50)
```

5.27.8 Blocks

Raster datasets are generally composed of multiple blocks of data and windowed reads and writes are most efficient when the windows match the dataset's own block structure. When a file is opened to read, the shape of blocks for any band can be had from the `block_shapes` property.

```
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     for i, shape in enumerate(src.block_shapes, 1):
...         print((i, shape))
...
(1, (3, 791))
(2, (3, 791))
(3, (3, 791))
```

The block windows themselves can be had from the `block_windows` function.

```
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     for ji, window in src.block_windows(1):
...         print((ji, window))
...
((0, 0), ((0, 3), (0, 791)))
((1, 0), ((3, 6), (0, 791)))
...
```

This function returns an iterator that yields a pair of values. The second is a window tuple that can be used in calls to `read()` or `write()`. The first is the pair of row and column indexes of this block within all blocks of the dataset.

You may read windows of data from a file block-by-block like this.

```
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     for ji, window in src.block_windows(1):
...         r = src.read(1, window=window)
...         print(r.shape)
...         break
...
(3, 791)
```

Well-bred files have identically blocked bands, but GDAL allows otherwise and it's a good idea to test this assumption in your code.

```
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     assert len(set(src.block_shapes)) == 1
...     for ji, window in src.block_windows(1):
...         b, g, r = (src.read(k, window=window) for k in (1, 2, 3))
...         print((ji, r.shape, g.shape, b.shape))
...         break
...
((0, 0), (3, 791), (3, 791), (3, 791))
```

The `block_shapes` property is a band-ordered list of block shapes and `set(src.block_shapes)` gives you the set of unique shapes. Asserting that there is only one item in the set is effectively the same as asserting that all bands have the same block structure. If they do, you can use the same windows for each.

5.28 Writing Datasets

Todo:

- appending to existing data
 - context manager
 - write 3d vs write 2d
 - document issues with writing compressed files (per #77)
 - **discuss and refer to topics**
 - creation options
 - transforms
 - dtypes
 - block windows
-

Opening a file in writing mode is a little more complicated than opening a text file in Python. The dimensions of the raster dataset, the data types, and the specific format must be specified.

Here's an example of basic rasterio functionality. An array is written to a new single band TIFF.

```
# Register GDAL format drivers and configuration options with a
# context manager.
with rasterio.Env():
```

(continues on next page)

(continued from previous page)

```
# Write an array as a raster band to a new 8-bit file. For
# the new file's profile, we start with the profile of the source
profile = src.profile

# And then change the band count to 1, set the
# dtype to uint8, and specify LZW compression.
profile.update(
    dtype=rasterio.uint8,
    count=1,
    compress='lzw')

with rasterio.open('example.tif', 'w', **profile) as dst:
    dst.write(array.astype(rasterio.uint8), 1)

# At the end of the ``with rasterio.Env()`` block, context
# manager exits and all drivers are de-registered.
```

Writing data mostly works as with a Python file. There are a few format- specific differences.

5.28.1 Supported Drivers

GTiff is the only driver that supports writing directly to disk. GeoTiffs use the `RasterUpdater` and leverage the full capabilities of the `GDALCreate()` function. We highly recommend using GeoTiff driver for writing as it is the best-tested and best-supported format.

Some other formats that are writable by GDAL can also be written by Rasterio. These use an `IndirectRasterUpdater` which does not create directly but uses a temporary in-memory dataset and `GDALCreateCopy()` to produce the final output.

Some formats are known to produce invalid results using the `IndirectRasterUpdater`. These formats will raise a `RasterioIOError` if you attempt to write to the. Currently this applies to the `netCDF` driver but please let us know if you experience problems writing other formats.

PYTHON API REFERENCE

6.1 rasterio package

6.1.1 Subpackages

rio CLI

6.1.2 Submodules

rasterio.control module

Ground control points

class rasterio.control.**GroundControlPoint**(*row=None, col=None, x=None, y=None, z=None, id=None, info=None*)

Bases: `object`

A mapping of row, col image coordinates to x, y, z.

asdict()

A dict representation of the GCP

rasterio.coords module

rasterio.crs module

rasterio.drivers module

rasterio.dtypes module

rasterio.enums module

rasterio.env module

Rasterio's GDAL/AWS environment

class rasterio.env.**Env**(*session=None, aws_unsigned=False, profile_name=None, session_class=<function Session.aws_or_dummy>, **options*)

Bases: `object`

Abstraction for GDAL and AWS configuration

The GDAL library is stateful: it has a registry of format drivers, an error stack, and dozens of configuration options.

Rasterio's approach to working with GDAL is to wrap all the state up using a Python context manager (see PEP 343, <https://www.python.org/dev/peps/pep-0343/>). When the context is entered GDAL drivers are registered, error handlers are configured, and configuration options are set. When the context is exited, drivers are removed from the registry and other configurations are removed.

Example

```
with rasterio.Env(GDAL_CACHEMAX=128000000) as env:
    # All drivers are registered, GDAL's raster block cache
    # size is set to 128 MB.
    # Commence processing...
    ...
    # End of processing.

# At this point, configuration options are set to their
# previous (possible unset) values.
```

A boto3 session or boto3 session constructor arguments `aws_access_key_id`, `aws_secret_access_key`, `aws_session_token` may be passed to Env's constructor. In the latter case, a session will be created as soon as needed. AWS credentials are configured for GDAL as needed.

aws_creds_from_context_options()

credentialize()

Get credentials and configure GDAL

Note well: this method is a no-op if the GDAL environment already has credentials, unless session is not None.

Return type

None

classmethod default_options()

Default configuration options

Parameters

None –

Return type

dict

drivers()

Return a mapping of registered drivers.

classmethod from_defaults(*args, **kwargs)

Create an environment with default config options

Parameters

- **args** (*optional*) – Positional arguments for Env()
- **kwargs** (*optional*) – Keyword arguments for Env()

Return type

Env

Notes

The items in kwargs will be overlaid on the default values.

class rasterio.env.GDALVersion(*major=0, minor=0*)

Bases: `object`

Convenience class for obtaining GDAL major and minor version components and comparing between versions. This is highly simplistic and assumes a very normal numbering scheme for versions and ignores everything except the major and minor components.

at_least(*other*)

major

minor

classmethod parse(*input*)

Parses input tuple or string to GDALVersion. If input is a GDALVersion instance, it is returned.

Parameters

input (*tuple* of (*major*, *minor*), *string*, or *instance* of `GDALVersion`) –

Return type

GDALVersion instance

classmethod runtime()

Return GDALVersion of current GDAL runtime

class rasterio.env.NullContextManager

Bases: `object`

class rasterio.env.ThreadEnv

Bases: `_local`

rasterio.env.defenv(***options*)

Create a default environment if necessary.

rasterio.env.delenv()

Delete options in the existing environment.

rasterio.env.ensure_env(*f*)

A decorator that ensures an env exists before a function calls any GDAL C functions.

rasterio.env.ensure_env_credentialled(*f*)

DEPRECATED alias for ensure_env_with_credentials

rasterio.env.ensure_env_with_credentials(*f*)

Ensures a config environment exists and is credentialized

Parameters

f (*function*) – A function.

Return type

A function wrapper.

Notes

The function wrapper checks the first argument of `f` and credentializes the environment if the first argument is a URI with scheme “s3”.

`rasterio.env.env_ctx_if_needed()`

Return an `Env` if one does not exist

Return type

`Env` or a do-nothing context manager

`rasterio.env.getenv()`

Get a mapping of current options.

`rasterio.env.hascreds()`

`rasterio.env.hasenv()`

`rasterio.env.require_gdal_version(version, param=None, values=None, is_max_version=False, reason=“”)`

A decorator that ensures the called function or parameters are supported by the runtime version of GDAL. Raises `GDALVersionError` if conditions are not met.

Examples

```
@require_gdal_version('2.2')
def some_func():
```

calling `some_func` with a runtime version of GDAL that is < 2.2 raises a `GDALVersionError`.

```
@require_gdal_version('2.2', param='foo')
def some_func(foo='bar'):
```

calling `some_func` with parameter `foo` of any value on GDAL < 2.2 raises a `GDALVersionError`.

```
@require_gdal_version('2.2', param='foo', values=('bar',))
def some_func(foo=None):
```

calling `some_func` with parameter `foo` and value `bar` on GDAL < 2.2 raises a `GDALVersionError`.

Parameters

- **version** (*tuple*, *string*, or `GDALVersion`) –
- **param** (*string* (optional, default: `None`)) – If *values* are absent, then all use of this parameter with a value other than default value requires at least GDAL *version*.
- **values** (*tuple*, *list*, or *set* (optional, default: `None`)) – contains values that require at least GDAL *version*. *param* is required for *values*.
- **is_max_version** (*bool* (optional, default: `False`)) – if `True` indicates that the version provided is the maximum version allowed, instead of requiring at least that version.
- **reason** (*string* (optional, default: `“”`)) – custom error message presented to user in addition to message about GDAL version. Use this to provide an explanation of what changed if necessary context to the user.

Return type

wrapped function

`rasterio.env.setenv(**options)`

Set options in the existing environment.

rasterio.errors module

Errors and Warnings.

exception `rasterio.errors.BandOverviewError`

Bases: `UserWarning`

Raised when a band overview access fails.

exception `rasterio.errors.CRSError`

Bases: `ValueError`

Raised when a CRS string or mapping is invalid or cannot serve to define a coordinate transformation.

exception `rasterio.errors.DatasetAttributeError`

Bases: `RasterioError`, `NotImplementedError`

Raised when dataset attributes are misused

exception `rasterio.errors.DatasetIOShapeError`

Bases: `RasterioError`

Raised when data buffer shape is a mismatch when reading and writing

exception `rasterio.errors.DriverCapabilityError`

Bases: `RasterioError`, `ValueError`

Raised when a format driver can't a feature such as writing.

exception `rasterio.errors.DriverRegistrationError`

Bases: `ValueError`

Raised when a format driver is requested but is not registered.

exception `rasterio.errors.EnvError`

Bases: `RasterioError`

Raised when the state of GDAL/AWS environment cannot be created or modified.

exception `rasterio.errors.FileOverwriteError(message)`

Bases: `FileError`

Raised when Rasterio's CLI refuses to clobber output files.

exception `rasterio.errors.GDALBehaviorChangeException`

Bases: `RuntimeError`

Raised when GDAL's behavior differs from the given arguments. For example, antimeridian cutting is always on as of GDAL 2.2.0. Users expecting it to be off will be presented with a MultiPolygon when the rest of their code expects a Polygon.

Examples

```
# Raises an exception on GDAL >= 2.2.0
rasterio.warp.transform_geometry(
    src_crs, dst_crs, antimeridian_cutting=False)
```

exception rasterio.errors.GDALOptionNotImplementedError

Bases: [RasterioError](#)

A dataset opening or dataset creation option can't be supported

This will be raised from Rasterio's shim modules. For example, when a user passes arguments to `open_dataset()` that can't be evaluated by GDAL 1.x.

exception rasterio.errors.GDALVersionError

Bases: [RasterioError](#)

Raised if the runtime version of GDAL does not meet the required version of GDAL.

exception rasterio.errors.InvalidArrayError

Bases: [RasterioError](#)

Raised when methods are passed invalid arrays

exception rasterio.errors.NodataShadowWarning

Bases: [UserWarning](#)

Warn that a dataset's nodata attribute is shadowing its alpha band.

exception rasterio.errors.NotGeoreferencedWarning

Bases: [UserWarning](#)

Warn that a dataset isn't georeferenced.

exception rasterio.errors.OverviewCreationError

Bases: [RasterioError](#)

Raised when creation of an overview fails

exception rasterio.errors.PathError

Bases: [RasterioError](#)

Raised when a dataset path is malformed or invalid

exception rasterio.errors.RPCError

Bases: [ValueError](#)

Raised when RPC transformation is invalid

exception rasterio.errors.RasterBlockError

Bases: [RasterioError](#)

Raised when raster block access fails

exception rasterio.errors.RasterioDeprecationWarning

Bases: [FutureWarning](#)

Rasterio module deprecations

Following <https://www.python.org/dev/peps/pep-0565/#additional-use-case-for-futurewarning> we base this on `FutureWarning` while continuing to support Python < 3.7.

exception `rasterio.errors.RasterioError`

Bases: `Exception`

Root exception class

exception `rasterio.errors.RasterioIOError`

Bases: `OSError`

Raised when a dataset cannot be opened using one of the registered format drivers.

exception `rasterio.errors.ResamplingAlgorithmError`

Bases: `RasterioError`

Raised when a resampling algorithm is invalid or inapplicable

exception `rasterio.errors.ShapeSkipWarning`

Bases: `UserWarning`

Warn that an invalid or empty shape in a collection has been skipped

exception `rasterio.errors.StatisticsError`

Bases: `RasterioError`

Raised when dataset statistics cannot be computed.

exception `rasterio.errors.TransformError`

Bases: `RasterioError`

Raised when transform arguments are invalid

exception `rasterio.errors.TransformWarning`

Bases: `UserWarning`

Warn that coordinate transformations may behave unexpectedly

exception `rasterio.errors.UnsupportedOperation`

Bases: `RasterioError`

Raised when reading from a file opened in 'w' mode

exception `rasterio.errors.WarpOperationError`

Bases: `RasterioError`

Raised when a warp operation fails.

exception `rasterio.errors.WarpOptionsError`

Bases: `RasterioError`

Raised when options for a warp operation are invalid

exception `rasterio.errors.WarpedVRTError`

Bases: `RasterioError`

Raised when WarpedVRT can't be initialized

exception `rasterio.errors.WindowError`

Bases: `RasterioError`

Raised when errors occur during window operations

exception `rasterio.errors.WindowEvaluationError`

Bases: `ValueError`

Raised when window evaluation fails

rasterio.features module

rasterio.fill module

rasterio.io module

rasterio.mask module

rasterio.merge module

rasterio.path module

rasterio.plot module

rasterio.profiles module

rasterio.rpc module

```
class rasterio.rpc.RPC(height_off, height_scale, lat_off, lat_scale, line_den_coeff, line_num_coeff, line_off,  
                      line_scale, long_off, long_scale, samp_den_coeff, samp_num_coeff, samp_off,  
                      samp_scale, err_bias=None, err_rand=None)
```

Bases: `object`

Rational Polynomial Coefficients used to map (x, y, z) <-> (row, col) coordinates.

This class contains a mapping between various RPC attributes and values.

err_bias, err_rand

The RMS bias and random error in meters per horizontal axis of all points in image.

Type

`float`, optional

lat_off, long_off, height_off

Geodetic latitude, longitude, and height offset.

Type

`float`

lat_scale, long_scale, height_scale

Geodetic latitude, longitude, and height scaling.

Type

`float`

line_off, samp_off

Line (row) and sample (column) offset.

Type

`float`

line_scale, samp_scale

Line (row) and sample (column) offset.

Type

`float`

line_num_coeff, line_den_coeff, samp_num_coeff, samp_den_coeff

The twenty coefficients describing a numerator or denominator polynomial corresponding to line (row) or sample (col).

Type

list

err_bias

err_rand

classmethod from_gdal(*rpcs*)

Deserialize dict values to float or list.

Return type

RPC

height_off

height_scale

lat_off

lat_scale

line_den_coeff

line_num_coeff

line_off

line_scale

long_off

long_scale

samp_den_coeff

samp_num_coeff

samp_off

samp_scale

to_dict()

Return a dictionary representation of RPC

to_gdal()

Serialize RPC attribute name and values in a form expected by GDAL.

Return type

dict

Notes

The *err_bias* and *err_rand* are optional, and are not written to datasets by GDAL.

rasterio.sample module

rasterio.session module

Abstraction for sessions in various clouds.

```
class rasterio.session.AWSSession(session=None, aws_unsigned=None, aws_access_key_id=None,
                                   aws_secret_access_key=None, aws_session_token=None,
                                   region_name=None, profile_name=None, endpoint_url=None,
                                   requester_pays=False)
```

Bases: *Session*

Configures access to secured resources stored in AWS S3.

```
static aws_or_dummy(*args, **kwargs)
```

Create an AWSSession if boto3 is available, else DummySession

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

```
static cls_from_path(path)
```

Find the session class suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.

Return type

class

property credentials

The session credentials as a dict

```
static from_environ(*args, **kwargs)
```

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static from_foreign_session(*session*, *cls=None*)

Create a session object matching the foreign *session*.

Parameters

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class*, *optional*) – The class to return.

Return type

Session

static from_path(*path*, **args*, ***kwargs*)

Create a session object suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

get_credential_options()

Get credentials as GDAL configuration options

Return type

dict

classmethod hascreds(*config*)

Determine if the given configuration has proper credentials

Parameters

- **cls** (*class*) – A Session class.
- **config** (*dict*) – GDAL configuration as a dict.

Return type

bool

class rasterio.session.AzureSession(*azure_storage_connection_string=None*,
azure_storage_account=None, *azure_storage_access_key=None*,
azure_unsigned=False)

Bases: *Session*

Configures access to secured resources stored in Microsoft Azure Blob Storage.

static aws_or_dummy(**args*, ***kwargs*)

Create an AWSSession if boto3 is available, else DummySession

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static `cls_from_path(path)`

Find the session class suited to the data at *path*.

Parameters

path (*str*) – A dataset path or identifier.

Return type

class

property `credentials`

The session credentials as a dict

static `from_environ(*args, **kwargs)`

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static `from_foreign_session(session, cls=None)`

Create a session object matching the foreign *session*.

Parameters

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class, optional*) – The class to return.

Return type

Session

static `from_path(path, *args, **kwargs)`

Create a session object suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

get_credential_options()

Get credentials as GDAL configuration options

Return type

dict

classmethod `hascreds(config)`

Determine if the given configuration has proper credentials

Parameters

- **cls** (*class*) – A Session class.

- **config** (*dict*) – GDAL configuration as a dict.

Return type*bool*

class rasterio.session.DummySession(*args, **kwargs)

Bases: *Session*

A dummy session.

credentials

The session credentials.

Type*dict*

static aws_or_dummy(*args, **kwargs)

Create an AWSSession if boto3 is available, else DummySession

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session*

static cls_from_path(path)

Find the session class suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.

Return type*class*

static from_environ(*args, **kwargs)

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session*

static from_foreign_session(session, cls=None)

Create a session object matching the foreign *session*.

Parameters

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class, optional*) – The class to return.

Return type*Session*

static from_path(*path*, *args, **kwargs)

Create a session object suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

get_credential_options()

Get credentials as GDAL configuration options

Return type

dict

classmethod hascreds(*config*)

Determine if the given configuration has proper credentials

Parameters

- **cls** (*class*) – A Session class.
- **config** (*dict*) – GDAL configuration as a dict.

Return type

bool

class rasterio.session.GSSession(*google_application_credentials=None*)

Bases: *Session*

Configures access to secured resources stored in Google Cloud Storage

static aws_or_dummy(*args, **kwargs)

Create an AWSSession if boto3 is available, else DummySession

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static cls_from_path(*path*)

Find the session class suited to the data at *path*.

Parameters

path (*str*) – A dataset path or identifier.

Return type

class

property credentials

The session credentials as a dict

static from_environ(*args, **kwargs)

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static from_foreign_session(session, cls=None)

Create a session object matching the foreign *session*.

Parameters

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class, optional*) – The class to return.

Return type

Session

static from_path(path, *args, **kwargs)

Create a session object suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

get_credential_options()

Get credentials as GDAL configuration options

Return type

dict

classmethod hascreds(config)

Determine if the given configuration has proper credentials

Parameters

- **cls** (*class*) – A Session class.
- **config** (*dict*) – GDAL configuration as a dict.

Return type

bool

class rasterio.session.OSSSession(oss_access_key_id=None, oss_secret_access_key=None, oss_endpoint=None)

Bases: *Session*

Configures access to secured resources stored in Alibaba Cloud OSS.

static `aws_or_dummy(*args, **kwargs)`

Create an `AWSSession` if `boto3` is available, else `DummySession`

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static `cls_from_path(path)`

Find the session class suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.

Return type

class

property credentials

The session credentials as a dict

static `from_environ(*args, **kwargs)`

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

static `from_foreign_session(session, cls=None)`

Create a session object matching the foreign *session*.

Parameters

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class, optional*) – The class to return.

Return type

Session

static `from_path(path, *args, **kwargs)`

Create a session object suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

Session

get_credential_options()

Get credentials as GDAL configuration options

Return type

`dict`

classmethod hascreds(*config*)

Determine if the given configuration has proper credentials

Parameters

- **cls** (*class*) – A Session class.
- **config** (*dict*) – GDAL configuration as a dict.

Return type

`bool`

class rasterio.session.Session

Bases: `object`

Base for classes that configure access to secured resources.

credentials

Keys and values for session credentials.

Type

`dict`

Notes

This class is not intended to be instantiated.

static aws_or_dummy(*args, **kwargs)

Create an AWSSession if boto3 is available, else DummySession

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type

`Session`

static cls_from_path(path)

Find the session class suited to the data at *path*.

Parameters

- **path** (*str*) – A dataset path or identifier.

Return type

`class`

static from_environ(*args, **kwargs)

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.

- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session***static from_foreign_session**(*session*, *cls=None*)Create a session object matching the foreign *session*.**Parameters**

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class, optional*) – The class to return.

Return type*Session***static from_path**(*path*, **args*, ***kwargs*)Create a session object suited to the data at *path*.**Parameters**

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session***get_credential_options**()

Get credentials as GDAL configuration options

Return type*dict***classmethod hascreds**(*config*)

Determine if the given configuration has proper credentials

Parameters

- **cls** (*class*) – A Session class.
- **config** (*dict*) – GDAL configuration as a dict.

Return type*bool***class rasterio.session.SwiftSession**(*session=None*, *swift_storage_url=None*, *swift_auth_token=None*, *swift_auth_v1_url=None*, *swift_user=None*, *swift_key=None*)Bases: *Session*

Configures access to secured resources stored in OpenStack Swift Object Storage.

static aws_or_dummy(**args*, ***kwargs*)

Create an AWSSession if boto3 is available, else DummySession

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.

- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session***static cls_from_path(path)**Find the session class suited to the data at *path*.**Parameters****path** (*str*) – A dataset path or identifier.**Return type**

class

property credentials

The session credentials as a dict

static from_environ(*args, **kwargs)

Create a session object suited to the environment.

Parameters

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session***static from_foreign_session(session, cls=None)**Create a session object matching the foreign *session*.**Parameters**

- **session** (*obj*) – A foreign session object.
- **cls** (*Session class, optional*) – The class to return.

Return type*Session***static from_path(path, *args, **kwargs)**Create a session object suited to the data at *path*.**Parameters**

- **path** (*str*) – A dataset path or identifier.
- **args** (*sequence*) – Positional arguments for the foreign session constructor.
- **kwargs** (*dict*) – Keyword arguments for the foreign session constructor.

Return type*Session***get_credential_options()**

Get credentials as GDAL configuration options :rtype: dict

classmethod hascreds(config)

Determine if the given configuration has proper credentials :param cls: A Session class. :type cls: class
 :param config: GDAL configuration as a dict. :type config: dict

Return type

bool

`rasterio.session.parse_bool(v)`

CPLTestBool equivalent

rasterio.shutil module

rasterio.tools module

rasterio.transform module

rasterio.vrt module

rasterio.warp module

rasterio.windows module

6.1.3 Module contents

CONTRIBUTING

Welcome to the Rasterio project. Here's how we work.

7.1 Code of Conduct

First of all: the Rasterio project has a code of conduct. Please read the `CODE_OF_CONDUCT.txt` file, it's important to all of us.

7.2 Rights

The BSD license (see `LICENSE.txt`) applies to all contributions.

7.3 Issue Conventions

The Rasterio issue tracker is for actionable issues.

Questions about installation, distribution, and usage should be taken to the project's [general discussion group](#). Opened issues which fall into one of these three categories may be perfunctorily closed.

Questions about development of Rasterio, brainstorming, requests for comment, and not-yet-actionable proposals are welcome in the project's [developers discussion group](#). Issues opened in Rasterio's GitHub repo which haven't been socialized there may be perfunctorily closed.

Rasterio is a relatively new project and highly active. We have bugs, both known and unknown.

Please search existing issues, open and closed, before creating a new one.

Rasterio employs C extension modules, so bug reports very often hinge on the following details:

- Operating system type and version (Windows? Ubuntu 20.04? 18.04?)
- The version and source of Rasterio (PyPI, Anaconda, or somewhere else?)
- The version and source of GDAL (UbuntuGIS? Homebrew?)

Please provide these details as well as tracebacks and relevant logs. When using the `$ rio` CLI logging can be enabled with `$ rio -v` and verbosity can be increased with `-vvv`. Short scripts and datasets demonstrating the issue are especially helpful!

7.4 Design Principles

Rasterio's API is both similar to and different from GDAL's API and this is intentional.

- Rasterio is a library for reading and writing raster datasets. Rasterio uses GDAL but is not a “Python binding for GDAL.”
- Rasterio aims to hide, or at least contain, the complexity of GDAL.
- Rasterio always prefers Python's built-in protocols and types or Numpy protocols and types over concepts from GDAL's data model.
- Rasterio keeps I/O separate from other operations. `rasterio.open()` is the only library function that operates on filenames and URIs. `dataset.read()`, `dataset.write()`, and their mask counterparts are the methods that perform I/O.
- Rasterio methods and functions should be free of side-effects and hidden inputs. This is challenging in practice because GDAL embraces global variables.
- Rasterio leans on analogies to other familiar Python APIs.

7.5 Dataset Objects

Our term for the kind of object that allows read and write access to raster data is *dataset object*. A dataset object might be an instance of *DatasetReader* or *DatasetWriter*. The canonical way to create a dataset object is by using the `rasterio.open()` function.

This is analogous to Python's use of [file object](#).

7.6 Path Objects

A *path object* specifies the name and address of a dataset within some space (filesystem, internet, cloud) along with optional parameters. The first positional argument of `rasterio.open()` is a path. Some path objects also have an *open* method which can be used to create a dataset object.

7.7 Band Objects

Unlike GDAL's original original data model, rasterio has no band objects. In this way it's more like GDAL's multi-dimensional API. A dataset's `read()` method returns N-D arrays.

7.8 GDAL Context

GDAL depends on some global context: a format driver registry, dataset connection pool, a raster block cache, a file header cache. Rasterio depends on this, too, but unlike GDAL's official Python bindings, delays initializing this context as long as possible and abstracts it with the help of a Python context manager.

7.9 Git Conventions

We use a variant of centralized workflow described in the [Git Book](#). Since Rasterio 1.0 we tag and release versions in the form: `x.y.z` version from the `maint-x.y` branch.

Work on features in a new branch of the mapbox/rasterio repo or in a branch on a fork. Create a [GitHub pull request](#) when the changes are ready for review. We recommend creating a pull request as early as possible to give other developers a heads up and to provide an opportunity for valuable early feedback.

7.10 Conventions

The `rasterio` namespace contains both Python and C extension modules. All C extension modules are written using [Cython](#). The Cython language is a superset of Python. Cython files end with `.pyx` and `.pxd` and are where we keep all the code that calls GDAL's C functions.

Rasterio works with Python versions 3.6 through 3.9.

We strongly prefer code adhering to [PEP8](#).

Tests are mandatory for new code. We use [pytest](#). Use pytest's parameterization feature.

We aspire to 100% coverage for Python modules but coverage of the Cython code is a future aspiration ([#515](#)).

Use [darker](#) to reformat code as you change it. We aren't going to run black on everything all at once.

Type hints are welcome as a part of refactoring work or new feature development. We aren't going to make a large initiative about adding hints to everything.

Changes should be noted in `CHANGES.txt`. New entries go above older entries.

7.11 New Containerized Development Environment

Rasterio has a new Dockerfile that can be used to create images and containers for exploring or testing the package.

The command `make dockertest` will build a Docker image based on one of the official GDAL images, start a container that mounts the working directory, and run `python setup.py develop && python -m pytest` in the container.

7.12 Historical Development Environment

If you prefer not to use the new development environment you may install rasterio's dependencies directly onto your computer.

Developing Rasterio requires Python 3.6 or any final release after and including 3.10. We prefer developing with the most recent version of Python but recognize this is not possible for all contributors. A C compiler is also required to leverage [existing protocols](#) for extending Python with C or C++. See the Windows install instructions in the [readme](#) for more information about building on Windows.

7.12.1 Initial Setup

First, clone Rasterio's git repo:

```
$ git clone https://github.com/rasterio/rasterio
```

Development should occur within a [virtual environment](#) to better isolate development work from custom environments.

In some cases installing a library with an accompanying executable inside a virtual environment causes the shell to initially look outside the environment for the executable. If this occurs try deactivating and reactivating the environment.

7.12.2 Installing GDAL

The GDAL library and its headers are required to build Rasterio. We do not have currently have guidance for any platforms other than Linux and OS X.

On Linux, GDAL and its headers should be available through your distro's package manager. For Ubuntu the commands are:

```
$ sudo add-apt-repository ppa:ubuntugis/ppa
$ sudo apt-get update
$ sudo apt-get install gdal-bin libgdal-dev
```

On OS X, Homebrew is a reliable way to get GDAL.

```
$ brew install gdal
```

7.12.3 Python build requirements

Provision a virtualenv with Rasterio's build requirements. Rasterio's `setup.py` script will not run unless Cython and Numpy are installed, so do this first from the Rasterio repo directory.

Linux users may need to install some additional Numpy dependencies:

```
$ sudo apt-get install libatlas-dev libatlas-base-dev gfortran
```

then:

```
$ pip install -U pip
$ pip install -r requirements-dev.txt
```

7.12.4 Installing Rasterio

Rasterio, its Cython extensions, normal dependencies, and dev dependencies can be installed with `$ pip`. Installing Rasterio in editable mode while developing is very convenient but only affects the Python files. Specifying the `[test]` extra in the command below tells `$ pip` to also install Rasterio's dev dependencies.

```
$ pip install -e .[test]
```

Any time a Cython (`.pyx` or `.pxd`) file is edited the extension modules need to be recompiled, which is most easily achieved with:

```
$ pip install -e .
```

When switching between Python versions the extension modules must be recompiled, which can be forced with `$ touch rasterio/*.pyx` and then re-installing with the command above. If this is not done an error claiming that an object has the wrong size, try recompiling is raised.

The dependencies required to build the docs can be installed with:

```
$ pip install -e .[docs]
```

7.12.5 Running the tests

Rasterio's tests live in `tests` <tests/> and generally match the main package layout.

To run the entire suite and the code coverage report:

Note: rasterio must be installed in editable mode in order to run tests.

```
$ python -m pytest --cov rasterio --cov-report term-missing
```

A single test file:

```
$ python -m pytest tests/test_band.py
```

A single test:

```
$ python -m pytest tests/test_band.py::test_band
```

7.12.6 Code Linting

flake8 is the code linter used by rasterio.

Using pre-commit to run flake8 code linting:

```
$ python -m pip install pre-commit
$ pre-commit install
```

7.13 Additional Information

More technical information lives on the wiki.

- <https://github.com/rasterio/rasterio/wiki/Development-Guide>
- <https://github.com/rasterio/rasterio/wiki/Exposing-GDAL-Functionality>
- <https://github.com/rasterio/rasterio/wiki/Cython-and-GDAL>

The long term goal is to consolidate into this document.

FREQUENTLY ASKED QUESTIONS

8.1 What does “RasterioIOError: file.ecw not recognized as a supported file format.” mean?

This exception is raised when none of rasterio’s format drivers can successfully open the specified dataset. In some cases it might be because the path is malformed, or the file is corrupted. Often, it is because your installation of rasterio does not provide the format driver. [ECW](#), for example, is an optional format driver and is not provided by the rasterio wheels in the Python Package Index. We’d like to keep the size of wheels to < ~20MB, and that means some GDAL features and format drivers must be left out. Other distribution channels for rasterio, such as conda-forge, may have different and larger sets of format drivers.

To see a list of the format drivers provided by your rasterio installation, run in your shell

```
rio env --formats
```

or, in a python interpreter,

```
with rasterio.Env() as env:
    print(list(env.drivers().items()))
```

8.2 Where is “ERROR 4: Unable to open EPSG support file gcs.csv” coming from and what does it mean?

The full message is “ERROR 4: Unable to open EPSG support file gcs.csv. Try setting the GDAL_DATA environment variable to point to the directory containing EPSG csv files.” The GDAL/OGR library prints this text to your process’s stdout stream when it can not find the gcs.csv data file it needs to interpret spatial reference system information stored with a dataset. If you’ve never seen this before, you can summon this message by setting GDAL_DATA to a bogus value in your shell and running a command like ogrinfo:

```
$ GDAL_DATA="/path/to/nowhere" ogrinfo example.shp -so example
INFO: Open of 'example.shp'
      using driver 'ESRI Shapefile' successful.

Layer name: example
Geometry: Polygon
Feature Count: 67
Extent: (-113.564247, 37.068981) - (-104.970871, 41.996277)
ERROR 4: Unable to open EPSG support file gcs.csv. Try setting the GDAL_DATA_
↪environment variable to point to the directory containing EPSG csv files.
```

If you're using GDAL software installed by a package management system like apt or yum, or Homebrew, or if you've built and installed it using `configure; make; make install`, you don't need to set the `GDAL_DATA` environment variable. That software has the right directory path built in. If you see this error, it's likely a sign that `GDAL_DATA` is set to a bogus value. Unset `GDAL_DATA` if it exists and see if that eliminates the error condition and the message.

Important: Activate your conda environments. The GDAL conda package will set `GDAL_DATA` to the proper value if you activate your conda environment. If you don't activate your conda environment, you are likely to see the error message shown above.

8.3 Why can't rasterio find proj.db (rasterio versions < 1.2.0)?

If you see `rasterio.errors.CRSError: The EPSG code is unknown. PROJ: proj_create_from_database: Cannot find proj.db` it is because the PROJ library (one of rasterio's dependencies) cannot find its database of projections and coordinate systems. In some installations the `PROJ_LIB` (`PROJ < 9.1`) | `PROJ_DATA` (`PROJ 9.1+`) environment variable must be set for PROJ to work properly.

Important: Activate your conda environments. The PROJ conda package will set `PROJ_LIB` (`PROJ < 9.1`) | `PROJ_DATA` (`PROJ 9.1+`) to the proper value if you activate your conda environment. If you don't activate your conda environment, you are likely to see the exception shown above.

8.4 Why can't rasterio find proj.db (rasterio from PyPI versions >= 1.2.0)?

Starting with version 1.2.0, rasterio wheels on PyPI include PROJ 7.x and GDAL 3.x. The libraries and modules in these wheels are incompatible with older versions of PROJ that may be installed on your system. If `PROJ_LIB` (`PROJ < 9.1`) | `PROJ_DATA` (`PROJ 9.1+`) is set in your program's environment and points to an older version of PROJ, you must unset this variable. Rasterio will then use the version of PROJ contained in the wheel.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

r

`rasterio.control`, [95](#)
`rasterio.env`, [95](#)
`rasterio.errors`, [99](#)
`rasterio.rpc`, [102](#)
`rasterio.session`, [104](#)

INDEX

A

`asdict()` (*rasterio.control.GroundControlPoint* method), 95
`at_least()` (*rasterio.env.GDALVersion* method), 97
`aws_creds_from_context_options()` (*rasterio.env.Env* method), 96
`aws_or_dummy()` (*rasterio.session.AWSSession* static method), 104
`aws_or_dummy()` (*rasterio.session.AzureSession* static method), 105
`aws_or_dummy()` (*rasterio.session.DummySession* static method), 107
`aws_or_dummy()` (*rasterio.session.GSSession* static method), 108
`aws_or_dummy()` (*rasterio.session.OSSSession* static method), 109
`aws_or_dummy()` (*rasterio.session.Session* static method), 111
`aws_or_dummy()` (*rasterio.session.SwiftSession* static method), 112
AWSSession (class in *rasterio.session*), 104
AzureSession (class in *rasterio.session*), 105

B

BandOverviewError, 99

C

`cls_from_path()` (*rasterio.session.AWSSession* static method), 104
`cls_from_path()` (*rasterio.session.AzureSession* static method), 105
`cls_from_path()` (*rasterio.session.DummySession* static method), 107
`cls_from_path()` (*rasterio.session.GSSession* static method), 108
`cls_from_path()` (*rasterio.session.OSSSession* static method), 110
`cls_from_path()` (*rasterio.session.Session* static method), 111
`cls_from_path()` (*rasterio.session.SwiftSession* static method), 113
`credentialize()` (*rasterio.env.Env* method), 96

`credentials` (*rasterio.session.AWSSession* property), 104
`credentials` (*rasterio.session.AzureSession* property), 106
`credentials` (*rasterio.session.DummySession* attribute), 107
`credentials` (*rasterio.session.GSSession* property), 108
`credentials` (*rasterio.session.OSSSession* property), 110
`credentials` (*rasterio.session.Session* attribute), 111
`credentials` (*rasterio.session.SwiftSession* property), 113
CRSError, 99

D

DatasetAttributeError, 99
DatasetIOShapeError, 99
`default_options()` (*rasterio.env.Env* class method), 96
`defenv()` (in module *rasterio.env*), 97
`delenv()` (in module *rasterio.env*), 97
DriverCapabilityError, 99
DriverRegistrationError, 99
`drivers()` (*rasterio.env.Env* method), 96
DummySession (class in *rasterio.session*), 107

E

`ensure_env()` (in module *rasterio.env*), 97
`ensure_env_credentialled()` (in module *rasterio.env*), 97
`ensure_env_with_credentials()` (in module *rasterio.env*), 97
Env (class in *rasterio.env*), 95
`env_ctx_if_needed()` (in module *rasterio.env*), 98
EnvError, 99
environment variable
 PROJ_DEBUG, 36
`err_bias` (*rasterio.rpc.RPC* attribute), 103
`err_rand` (*rasterio.rpc.RPC* attribute), 103

F

FileOverwriteError, 99

- [from_defaults\(\)](#) (*rasterio.env.Env* class method), 96
[from_environ\(\)](#) (*rasterio.session.AWSSession* static method), 104
[from_environ\(\)](#) (*rasterio.session.AzureSession* static method), 106
[from_environ\(\)](#) (*rasterio.session.DummySession* static method), 107
[from_environ\(\)](#) (*rasterio.session.GSSession* static method), 108
[from_environ\(\)](#) (*rasterio.session.OSSSession* static method), 110
[from_environ\(\)](#) (*rasterio.session.Session* static method), 111
[from_environ\(\)](#) (*rasterio.session.SwiftSession* static method), 113
[from_foreign_session\(\)](#) (*rasterio.session.AWSSession* static method), 104
[from_foreign_session\(\)](#) (*rasterio.session.AzureSession* static method), 106
[from_foreign_session\(\)](#) (*rasterio.session.DummySession* static method), 107
[from_foreign_session\(\)](#) (*rasterio.session.GSSession* static method), 109
[from_foreign_session\(\)](#) (*rasterio.session.OSSSession* static method), 110
[from_foreign_session\(\)](#) (*rasterio.session.Session* static method), 112
[from_foreign_session\(\)](#) (*rasterio.session.SwiftSession* static method), 113
[from_gdal\(\)](#) (*rasterio.rpc.RPC* class method), 103
[from_path\(\)](#) (*rasterio.session.AWSSession* static method), 105
[from_path\(\)](#) (*rasterio.session.AzureSession* static method), 106
[from_path\(\)](#) (*rasterio.session.DummySession* static method), 107
[from_path\(\)](#) (*rasterio.session.GSSession* static method), 109
[from_path\(\)](#) (*rasterio.session.OSSSession* static method), 110
[from_path\(\)](#) (*rasterio.session.Session* static method), 112
[from_path\(\)](#) (*rasterio.session.SwiftSession* static method), 113
- ## G
- [GDALBehaviorChangeException](#), 99
[GDALOptionNotImplementedError](#), 100
[GDALVersion](#) (class in *rasterio.env*), 97
[GDALVersionError](#), 100
- ## H
- [get_credential_options\(\)](#) (*rasterio.session.AWSSession* method), 105
[get_credential_options\(\)](#) (*rasterio.session.AzureSession* method), 106
[get_credential_options\(\)](#) (*rasterio.session.DummySession* method), 108
[get_credential_options\(\)](#) (*rasterio.session.GSSession* method), 109
[get_credential_options\(\)](#) (*rasterio.session.OSSSession* method), 110
[get_credential_options\(\)](#) (*rasterio.session.Session* method), 112
[get_credential_options\(\)](#) (*rasterio.session.SwiftSession* method), 113
[getenv\(\)](#) (in module *rasterio.env*), 98
[GroundControlPoint](#) (class in *rasterio.control*), 95
[GSSession](#) (class in *rasterio.session*), 108
- ## I
- [InvalidArrayError](#), 100
- ## L
- [lat_off](#) (*rasterio.rpc.RPC* attribute), 103
[lat_scale](#) (*rasterio.rpc.RPC* attribute), 103
[line_den_coeff](#) (*rasterio.rpc.RPC* attribute), 103
[line_num_coeff](#) (*rasterio.rpc.RPC* attribute), 103
[line_off](#) (*rasterio.rpc.RPC* attribute), 103
[line_scale](#) (*rasterio.rpc.RPC* attribute), 103
[long_off](#) (*rasterio.rpc.RPC* attribute), 103
[long_scale](#) (*rasterio.rpc.RPC* attribute), 103
- ## M
- [major](#) (*rasterio.env.GDALVersion* attribute), 97
[minor](#) (*rasterio.env.GDALVersion* attribute), 97

module

[rasterio.control](#), 95
[rasterio.env](#), 95
[rasterio.errors](#), 99
[rasterio.rpc](#), 102
[rasterio.session](#), 104

N

[NodataShadowWarning](#), 100
[NotGeoreferencedWarning](#), 100
[NullContextManager](#) (*class in rasterio.env*), 97

O

[OSSSession](#) (*class in rasterio.session*), 109
[OverviewCreationError](#), 100

P

[parse\(\)](#) (*rasterio.env.GDALVersion class method*), 97
[parse_bool\(\)](#) (*in module rasterio.session*), 114
[PathError](#), 100
[PROJ_DEBUG](#), 36

R

[RasterBlockError](#), 100
[rasterio.control](#)
 module, 95
[rasterio.env](#)
 module, 95
[rasterio.errors](#)
 module, 99
[rasterio.rpc](#)
 module, 102
[rasterio.session](#)
 module, 104
[RasterioDeprecationWarning](#), 100
[RasterioError](#), 100
[RasterioIOError](#), 101
[require_gdal_version\(\)](#) (*in module rasterio.env*), 98
[ResamplingAlgorithmError](#), 101
[RPC](#) (*class in rasterio.rpc*), 102
[RPCError](#), 100
[runtime\(\)](#) (*rasterio.env.GDALVersion class method*), 97

S

[samp_den_coeff](#) (*rasterio.rpc.RPC attribute*), 103
[samp_num_coeff](#) (*rasterio.rpc.RPC attribute*), 103
[samp_off](#) (*rasterio.rpc.RPC attribute*), 103
[samp_scale](#) (*rasterio.rpc.RPC attribute*), 103
[Session](#) (*class in rasterio.session*), 111
[setenv\(\)](#) (*in module rasterio.env*), 98
[ShapeSkipWarning](#), 101
[StatisticsError](#), 101
[SwiftSession](#) (*class in rasterio.session*), 112

T

[ThreadEnv](#) (*class in rasterio.env*), 97
[to_dict\(\)](#) (*rasterio.rpc.RPC method*), 103
[to_gdal\(\)](#) (*rasterio.rpc.RPC method*), 103
[TransformError](#), 101
[TransformWarning](#), 101

U

[UnsupportedOperation](#), 101

W

[WarpedVRTError](#), 101
[WarpOperationError](#), 101
[WarpOptionsError](#), 101
[WindowError](#), 101
[WindowEvaluationError](#), 101