rasterio Documentation

Sean Gillies

Apr 28, 2021

CONTENTS

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 2.7 and 3.3 or higher.

CHAPTER

ONE

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

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

Rasterio has one C library dependency: `GDAL >=1.11`. GDAL itself depends on many of other libraries provided by most major operating systems and also depends on the non standard GEOS and PROJ4 libraries.

Python package dependencies (see also requirements.txt): `affine, cligj, click, enum34, numpy`.

Development also requires (see requirements-dev.txt) Cython and other packages.

2.2 Installing from binaries

2.2.1 OS X

Binary wheels with the GDAL, GEOS, and PROJ4 libraries included are available for OS X versions 10.7+ starting with Rasterio version 0.17. To install, run `pip install rasterio`. These binary wheels are preferred by newer versions of pip. If you don't want these wheels and want to install from a source distribution, run `pip install rasterio --no-binary` instead.

The included GDAL library is fairly minimal, providing only the format drivers that ship with GDAL and are enabled by default. To get access to more formats, you must build from a source distribution (see below).

Binary wheels for other operating systems will be available in a future release.

2.2.2 Windows

Binary wheels for rasterio and GDAL are created by Christoph Gohlke and are available from his website.

To install rasterio, download both binaries for your system (rasterio and GDAL) and run something like this from the downloads folder:

```
$ pip install -U pip
$ pip install GDAL-1.11.2-cp27-none-win32.whl
$ pip install rasterio-0.24.0-cp27-none-win32.whl
```

2.3 Installing with Anaconda

To install rasterio on the Anaconda Python distribution, please visit the rasterio conda-forge page for install instructions. This build is maintained separately from the rasterio distribution on PyPi and packaging issues should be addressed on the rasterio conda-forge issue tracker.

2.4 Installing from the source distribution

Rasterio is a Python C extension and to build you'll need a working compiler (XCode on OS X etc). You'll also need Numpy preinstalled; the Numpy headers are required to run the rasterio setup script. Numpy has to be installed (via the indicated requirements file) before rasterio can be installed. See rasterio's Travis configuration for more guidance.

2.4.1 Linux

The following commands are adapted from Rasterio's Travis-CI configuration.

```
$ sudo add-apt-repository ppa:ubuntugis/ppa
$ sudo apt-get update
$ sudo apt-get install python-numpy gdal-bin libgdal-dev
$ pip install rasterio
```

Adapt them as necessary for your Linux system.

2.4.2 OS X

For a Homebrew based Python environment, do the following.

```
$ brew install gdal
$ pip install rasterio
```

2.4.3 Windows

You can download a binary distribution of GDAL from here. You will also need to download the compiled libraries and headers (include files).

When building from source on Windows, it is important to know that setup.py cannot rely on gdal-config, which is only present on UNIX systems, to discover the locations of header files and libraries that rasterio needs to compile its C extensions. On Windows, these paths need to be provided by the user. You will need to find the include files and the library files for gdal and use setup.py as follows.

```
$ python setup.py build_ext -I<path to gdal include files> -lgdal_i -L<path to gdal␣
↪library> install
```

With pip

```
$ pip install --no-use-pep517 --global-option -I<path to gdal include files> -lgdal_i␣
↪-L<path to gdal library> .
```

Note: `--no-use-pep517` is required as pip currently hasn't implemented a way for optional arguments to be passed to the build backend when using PEP 517. See here. for more details.

Alternatively environment variables (e.g. INCLUDE and LINK) used by MSVC compiler can be used to point to include directories and library files.

We have had success compiling code using the same version of Microsoft's Visual Studio used to compile the targeted version of Python (more info on versions used here.).

Note: The GDAL dll (gdal111.dll) and gdal-data directory need to be in your Windows PATH otherwise rasterio will fail to work.

CHAPTER

THREE

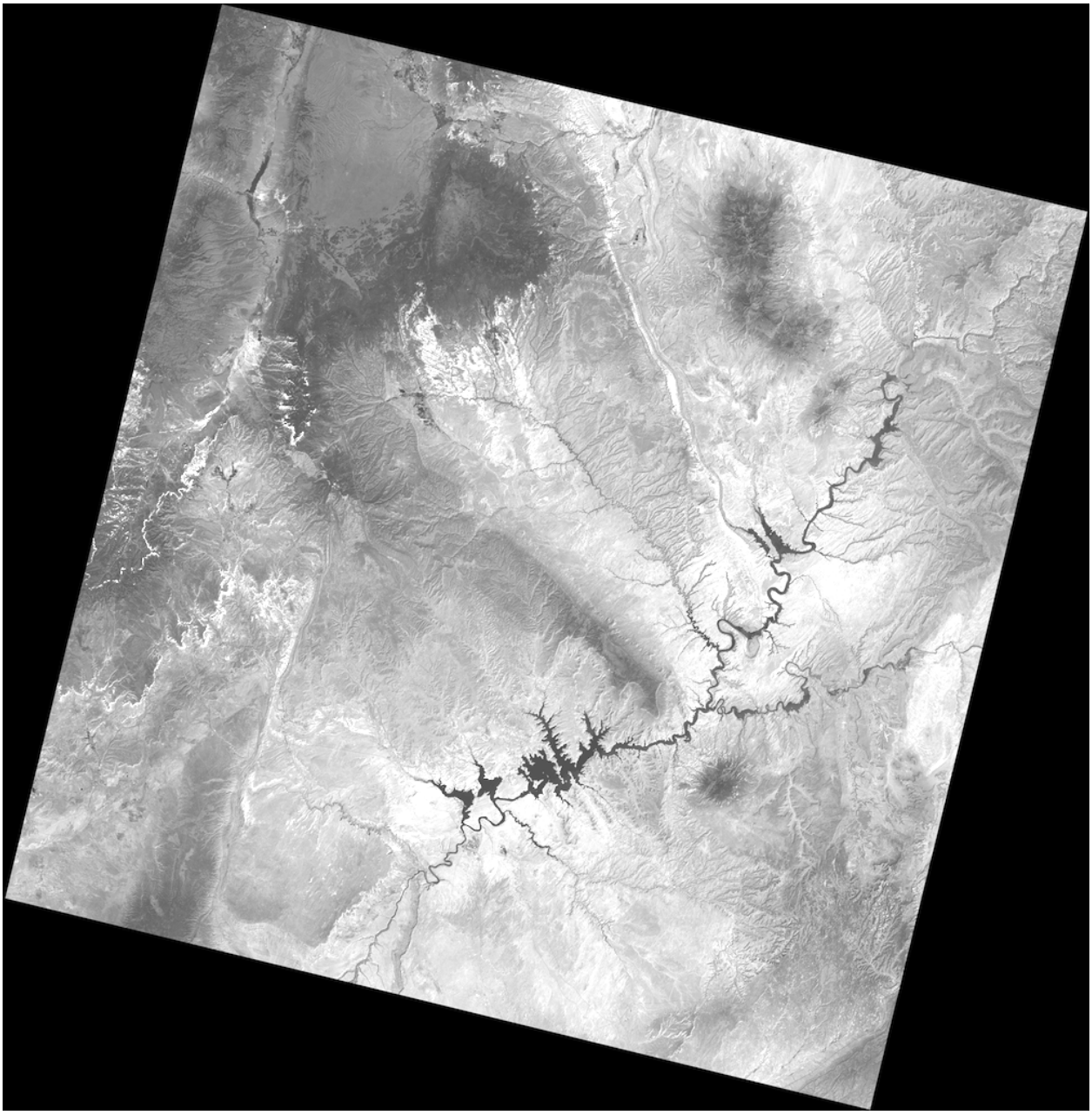
PYTHON QUICKSTART

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 Plateau[1]. 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
'example.tif'
```

(continues on next page)

(continued from previous page)

```
>>> 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 *indexes* and *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 *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 *transform* is an affine transformation matrix that maps pixel locations in (row, col) coordinates to (x, y) spatial positions. The product of this matrix and `(0, 0)`, the row and column 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 *crs* attribute and `transform` the georeferencing of a raster dataset is described and the dataset can 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 *read()* method returns a Numpy N-D array.

```
>>> 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 *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 *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)
```

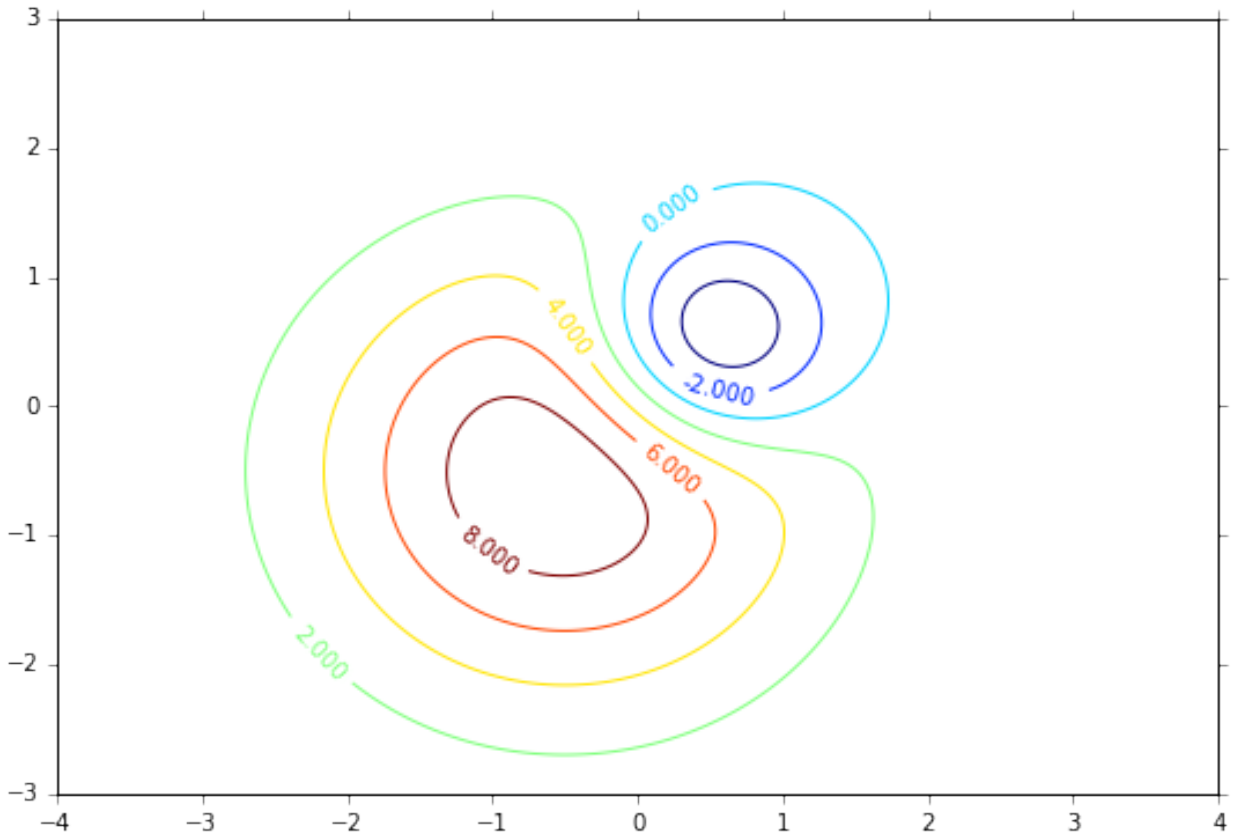
3.6 Creating data

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, -4.016666666666666,
       0.0, 0.03333333333333333, -3.016666666666666)
```

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 `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 *`write()`* method with the grid and target band number as arguments.

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

Then call the *`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,
) 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.

CHAPTER

FOUR

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
  --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.
  rasterize  Rasterize features.
  rm         Delete a dataset.
  sample     Sample a 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)

```
  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 bounds

Added in 0.10.

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
            ]
          ]
```

(continues on next page)

(continued from previous page)

```
        ],
        "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`.

4.3 calc

Added in 0.19

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.

4.4 clip

Added in 0.27

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)
```

4.5 convert

Added in 0.25

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*.

4.6 edit-info

Added in 0.24

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.

4.7 info

Added in 0.13

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,
      "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
}
```

4.8 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
```

(continues on next page)

(continued from previous page)

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

4.9 mask

Added in 0.21

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
```

4.10 merge

Added in 0.12.1

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

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

4.11 overview

New in 0.25

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
```

4.12 rasterize

New in 0.18.

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
```

4.13 rm

New in 1.0

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.

4.14 sample

New in 0.18.

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.

4.15 shapes

New in 0.11.

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 > shade.geojson
```

The resulting file, uploaded to Mapbox, looks like this: sgillies.j1ho338j.

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

```
$ rio shapes --mask --precision 6 tests/data/RGB.byte.tif | geojsonio
```

See http://bl.ocks.org/anonymous/raw/ef244954b719dba97926/.

4.16 stack

New in 0.15.

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*.

4.17 transform

New in 0.10.

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]
```

4.18 warp

New in 0.25

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
```

4.19 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.

4.20 Other commands?

Suggestions for other commands are welcome!

CHAPTER

FIVE

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 arithemtic 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 interpretted 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),
                255: (0, 0, 255, 255) })
        cmap = dst.colormap(1)
        # True
```

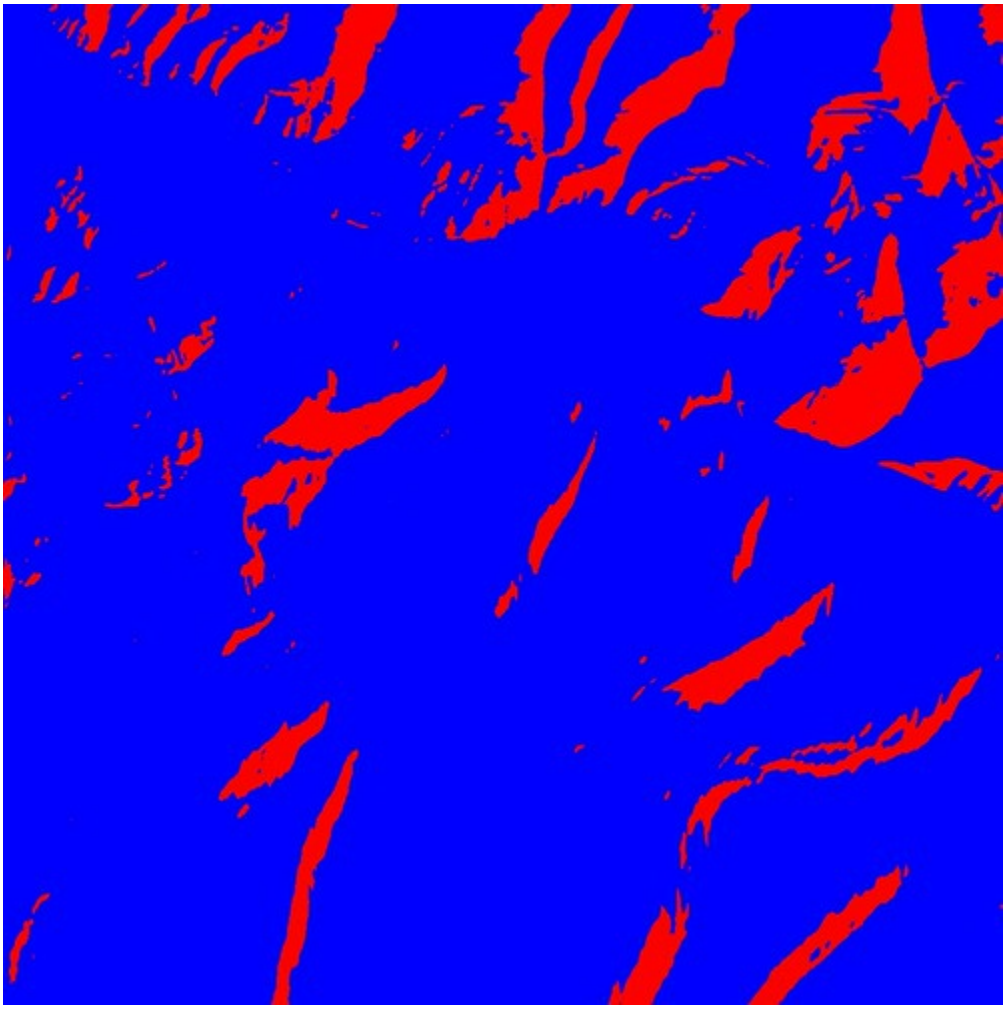
(continues on next page)

(continued from previous page)

```
        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 accross 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
                    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:
                    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
```

(continues on next page)

(continued from previous page)

```
            # 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 the `ThreadPoolExecutor` from Python 3's `concurrent.futures` module. 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/Writer cannot be shared by multiple processes, which means that each process needs to open the file seperately, 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 explicity 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 his 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 identifed 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

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)

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.

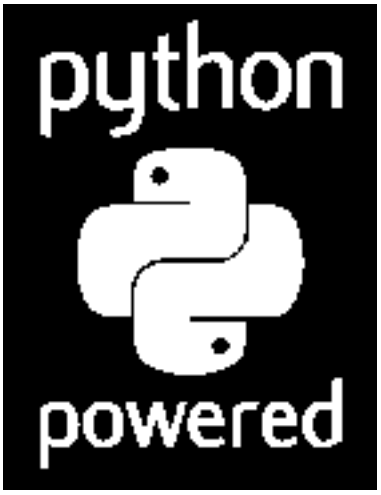
Again, to burn in georeferenced shapes, pass an appropriate transform for the image to be created.

```
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

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 `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 the CRS class's `wkt` attribute.

```
>>> src.crs.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

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 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'`.:

```
| x' |   | a b c | | x |
| y' | = | d e f | | y |
| 1  |   | 0 0 1 | | 1 |
```

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.

5.9.3 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.

5.9.4 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 https://gdal.org/api/gdal_alg.html#_CPPv424GDALCreateRPCTransformerP11GDALRPCInfoidPPc

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 programatically.

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.

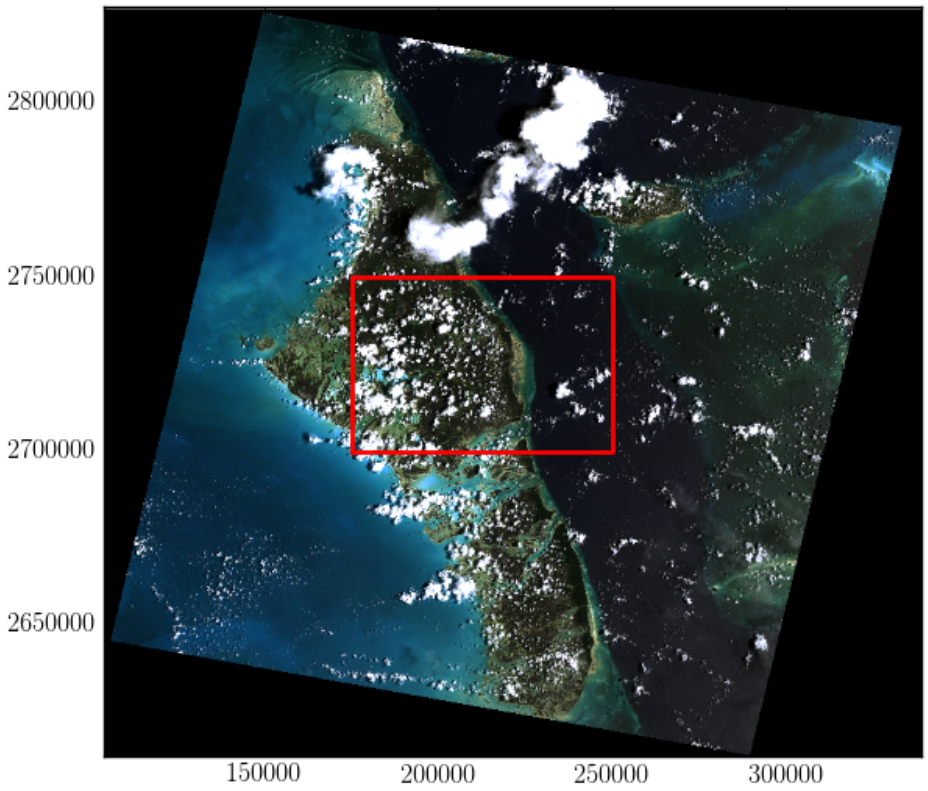
```
import fiona
import rasterio
import rasterio.mask

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
```

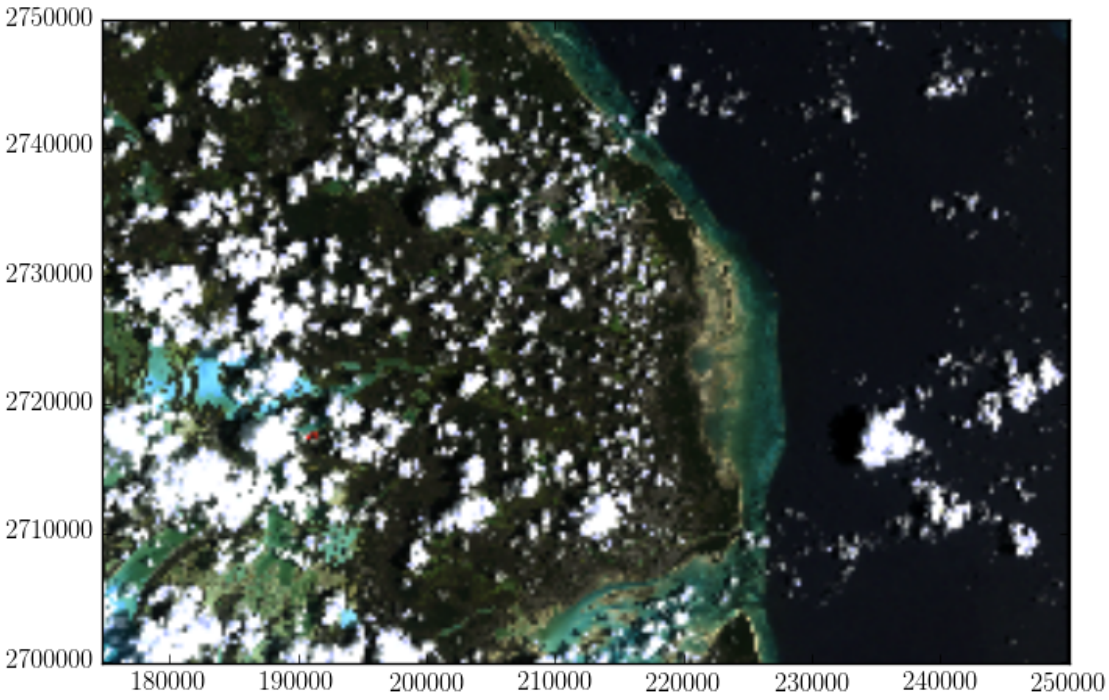
Using `plot` and `imshow` from `matplotlib`, we can see the region defined by the shapefile in red overlaid on the original raster.



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)
```



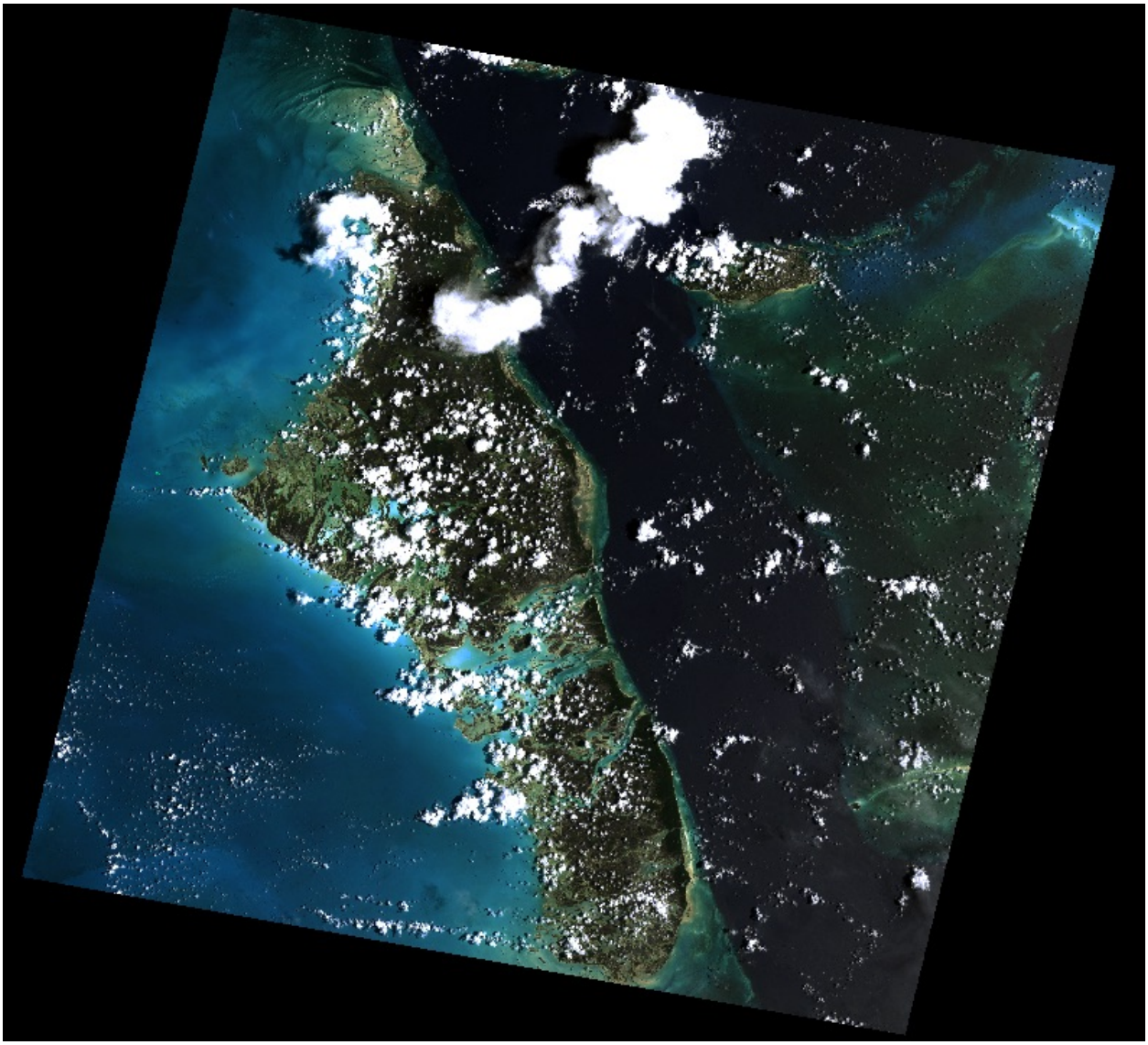
5.13 Nodata Masks

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 Numpy's masked array 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)
>>> src.nodata
0.0
```

5.13.1 Reading dataset masks

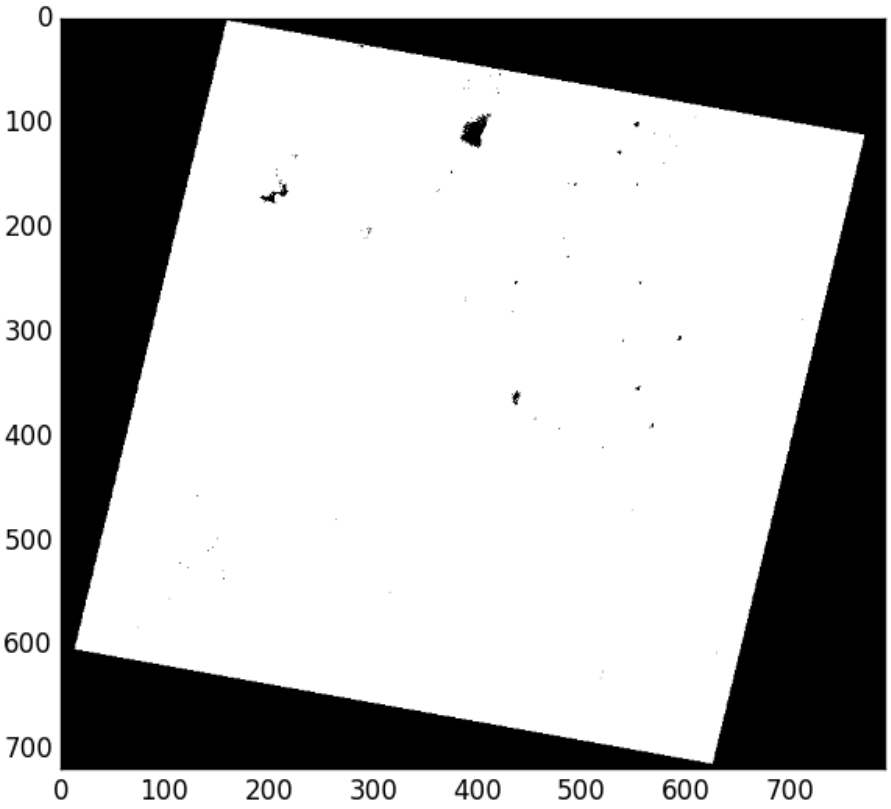
For every band of a dataset there is a mask. These masks can be had as arrays using the dataset's *read_masks()*` method. Below, `msk` is the valid data mask corresponding to the first dataset band.

```
>>> 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's *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.

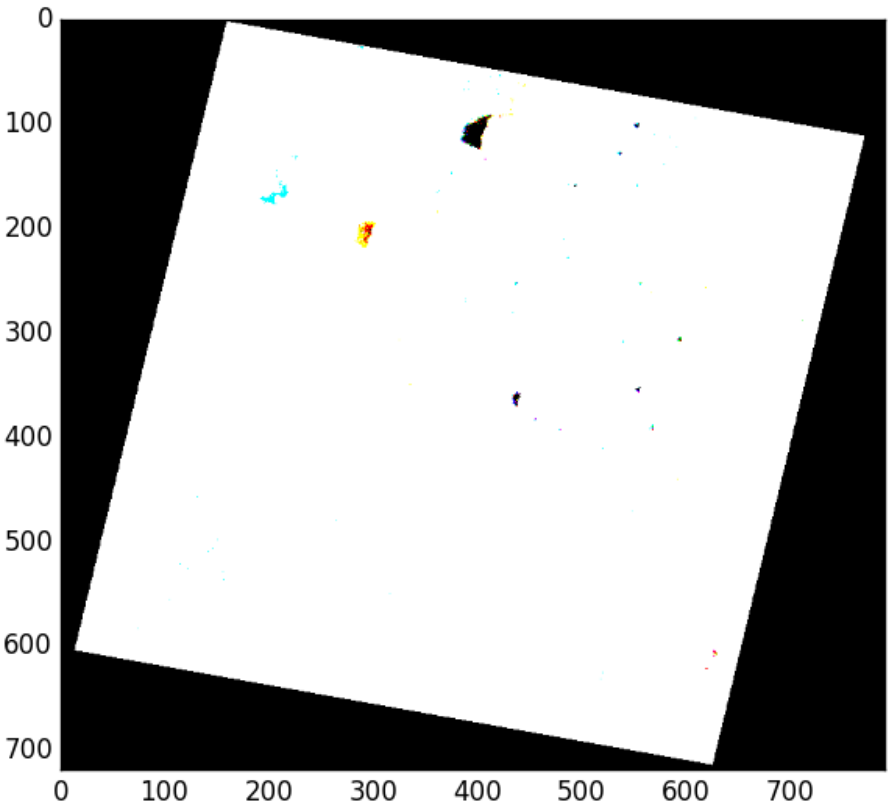
```
$ ls -l copy.tif*
-rw-r--r--@ 1 sean  staff  1713704 Mar 24 14:19 copy.tif
-rw-r--r--  1 sean  staff      916 Mar 24 14:25 copy.tif.msk
```

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+")
```

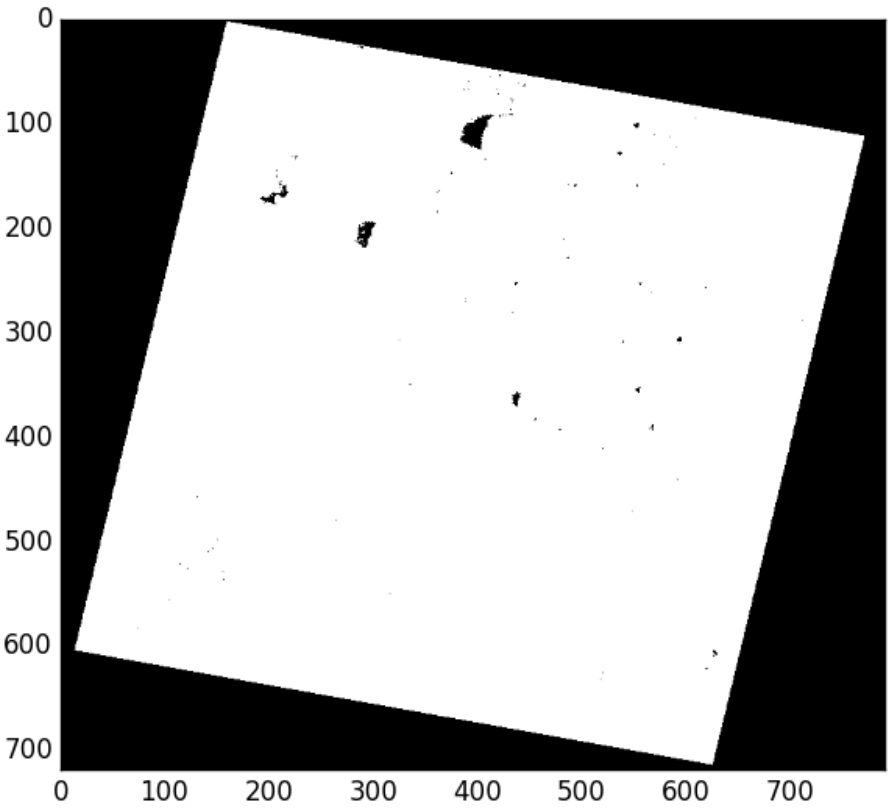
This time we'll read all 3 band masks (based on the nodata values, not a .msk GeoTIFF) and show them as an RGB image (with the help of *numpy.dstack()*):

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



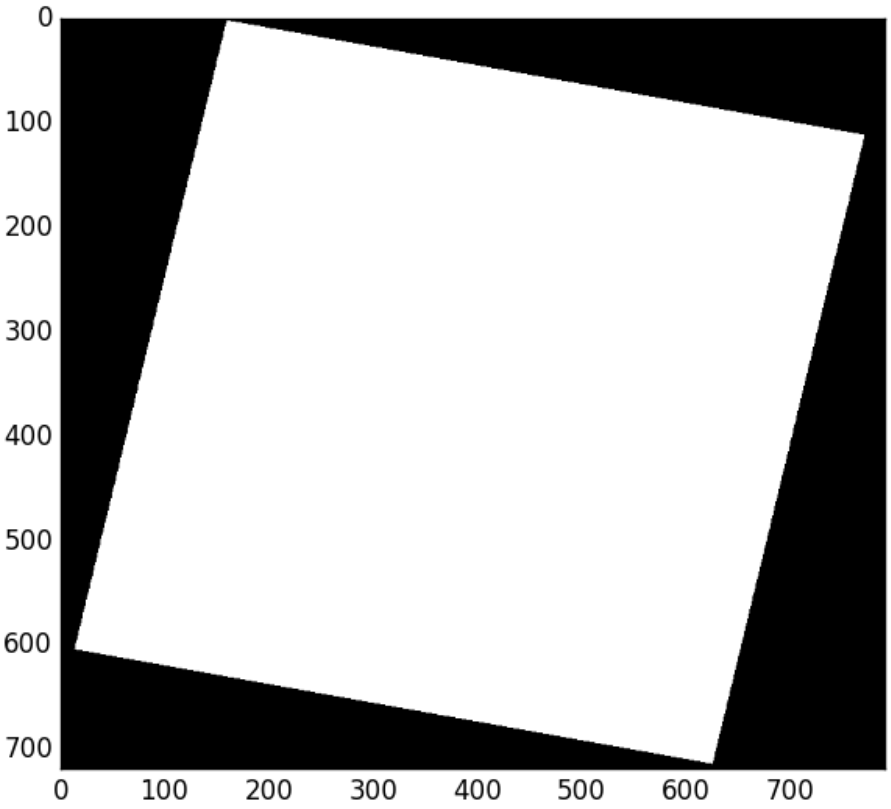
Colored regions appear where valid data pixels don't quite coincide. This is, again, an artifact of scaling data down to 8 bits per band. We'll begin by constructing a new mask array from the logical conjunction of the three band masks we've read.

```
>>> 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 numpy masked arrays.

```
>>> 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 `src.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 `src.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 explictly 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 alogrithms 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'
```

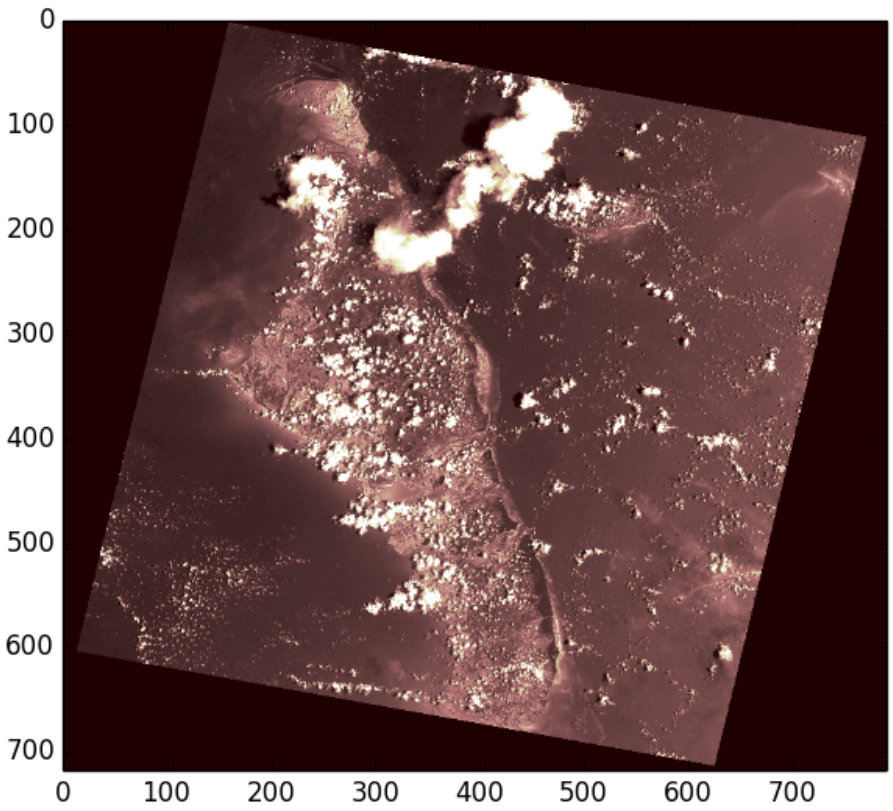
And to leverage the overviews, we can perform a decimated read at a reduced resolution which should allow libgdal to read directly from the overviews rather than compute them on-the-fly.

```
>>> 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()
```



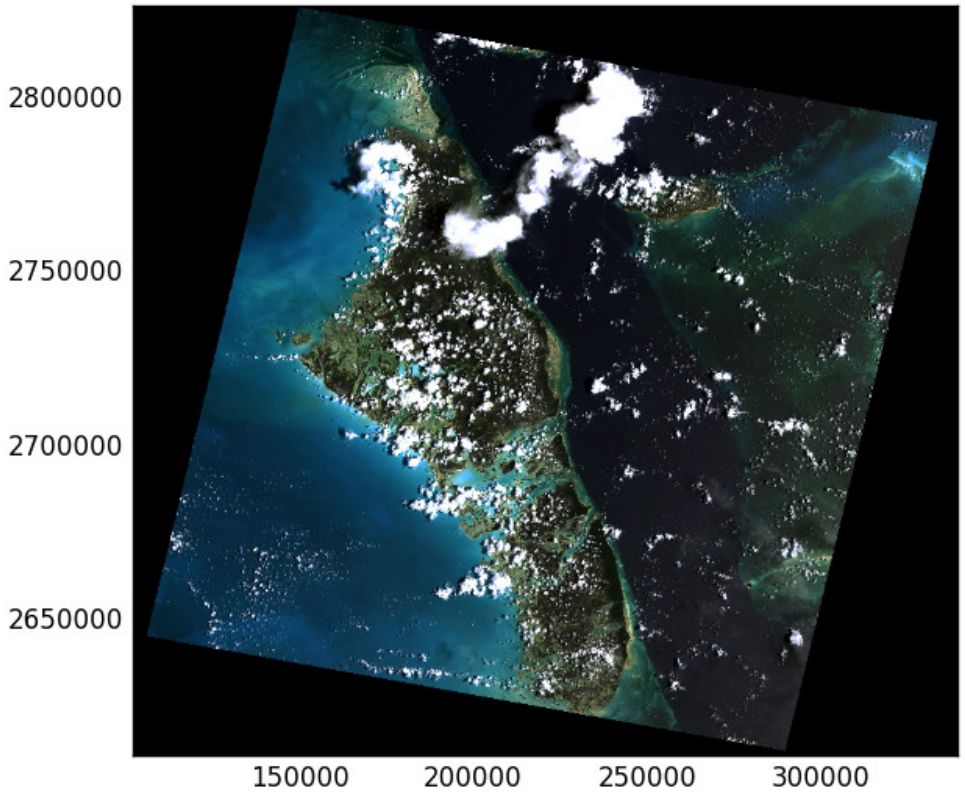
Rasterio also provides *`rasterio.plot.show()`* to perform common tasks such as displaying multi-band images as RGB and labeling the axes with proper geo-referenced extents.

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.

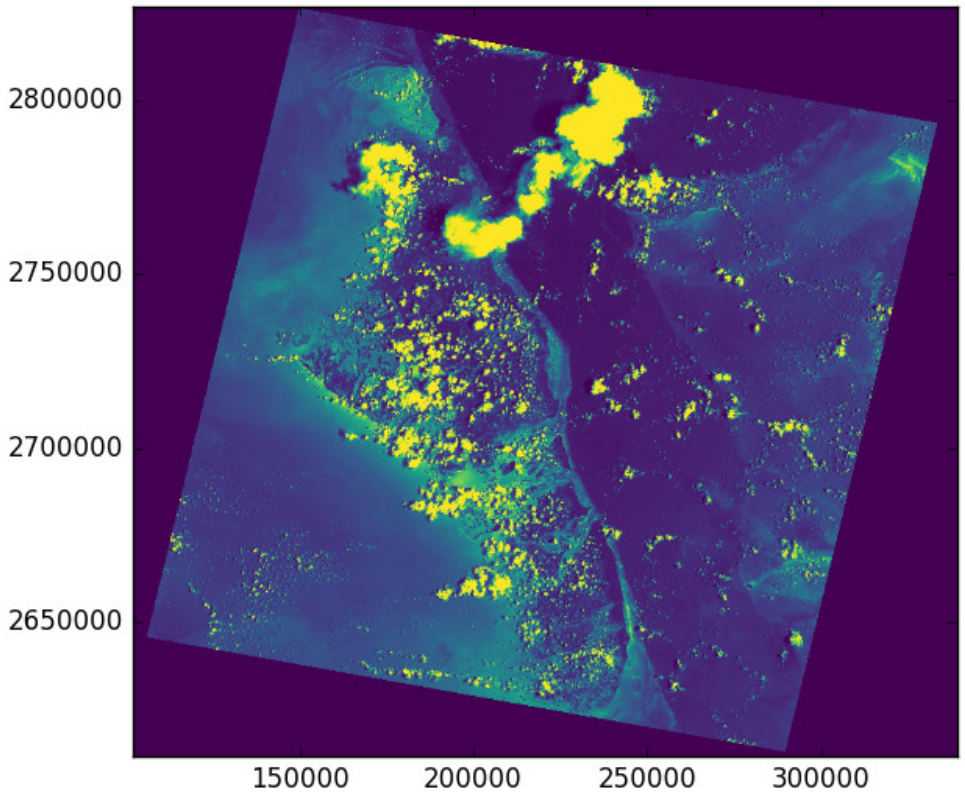
Thus the following operations for 3-band RGB data are equivalent. Note that when passing arrays, you can pass in a transform in order to get extent labels.

```
>>> 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...>
```



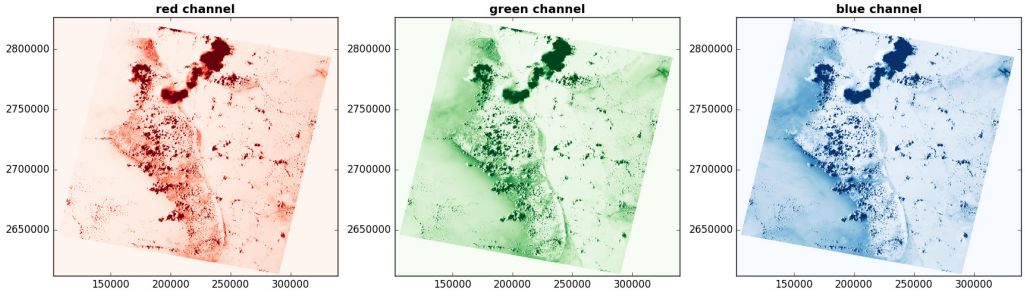
and similarly for single band plots. Note that you can pass in `cmap` to specify a matplotlib color ramp. Any kwargs passed to *`show()`* will be passed through to the underlying pyplot functions.

```
>>> 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...>
```



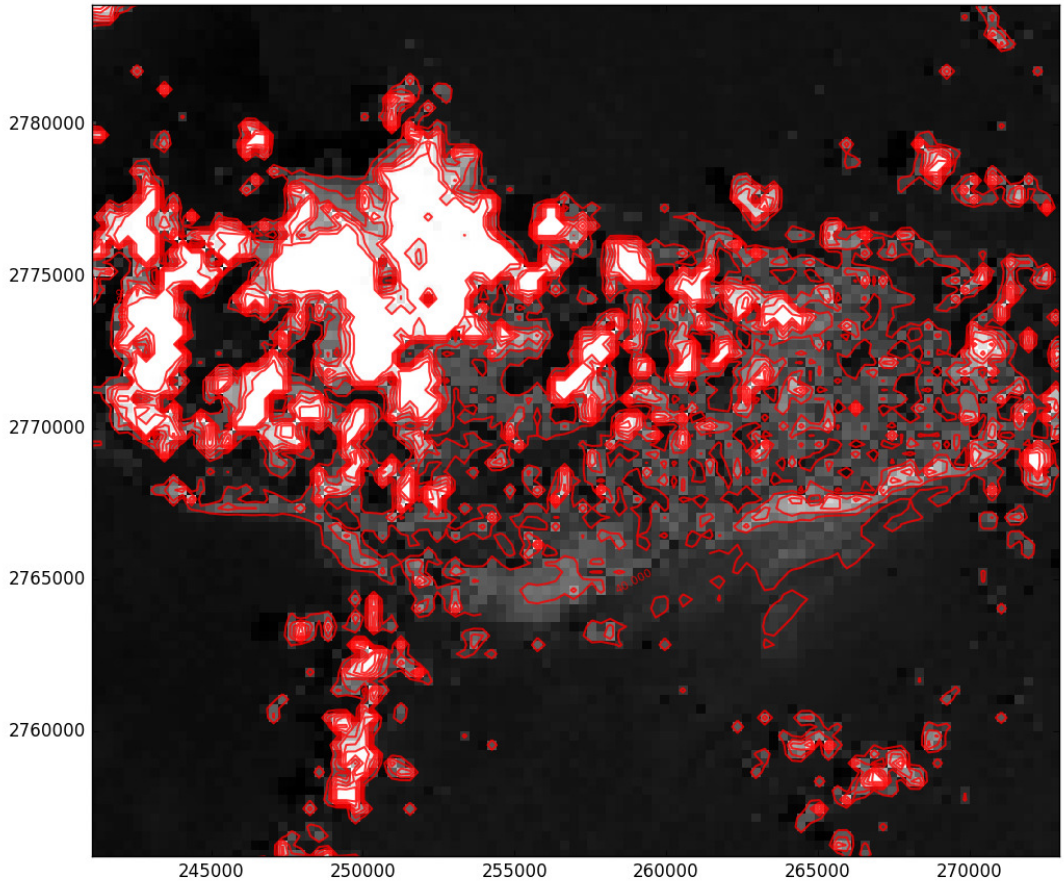
You can create a figure with multiple subplots by passing the `show(..., ax=ax1)` argument. Also note that this example demonstrates setting the overall figure size and sets a title for each subplot.

```
>>> 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()
```



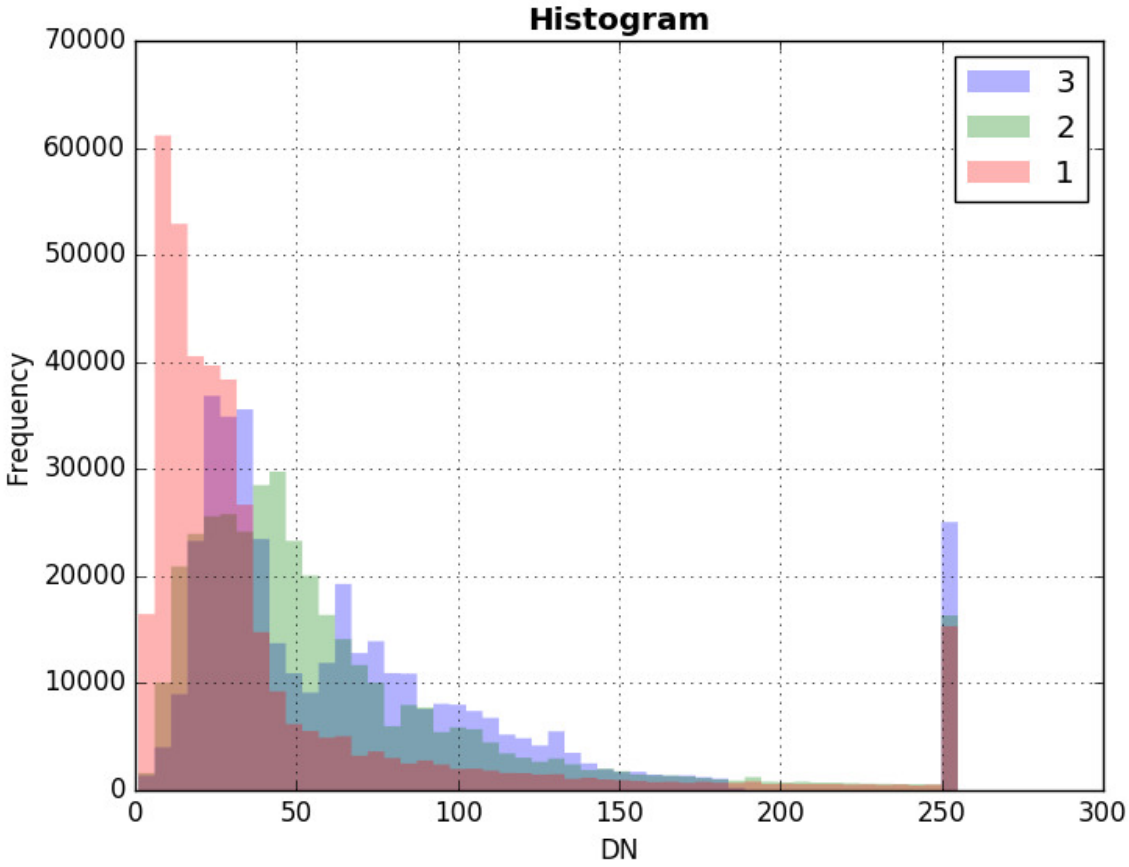
For single-band rasters, there is also an option to generate contours.

```
>>> 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()
```



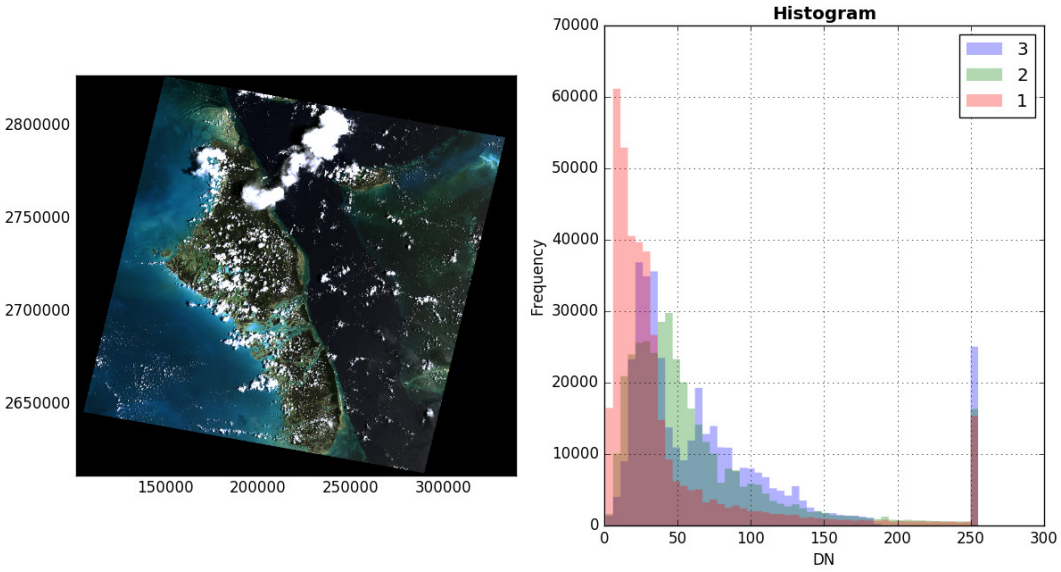
Rasterio also provides a *show_hist()* function for generating histograms of single or multiband rasters:

```
>>> 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'

    # Add GeoTIFF-specific keyword arguments.
```

(continues on next page)

(continued from previous page)

```
    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

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.

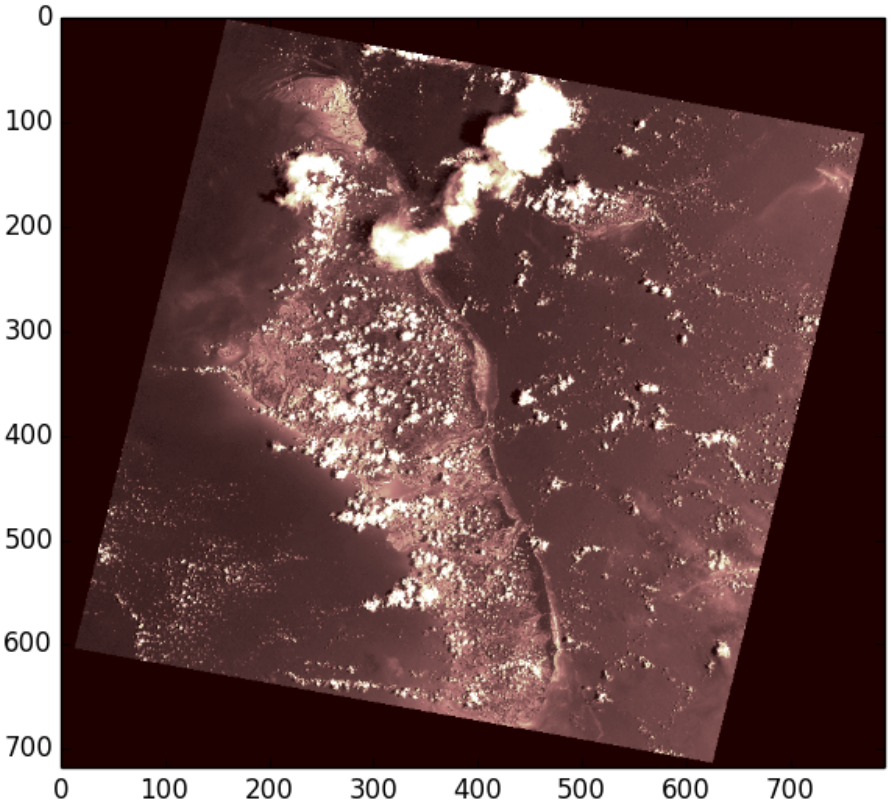
```
>>> open('/lol/wut.tif')
Traceback (most recent call last):
  ...
FileNotFoundError: [Errno 2] No such file or directory: '/lol/wut.tif'
>>> rasterio.open('/lol/wut.tif')
Traceback (most recent call last):
  ...
rasterio.errors.RasterioIOError: No such file or directory
```

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-dimensonal 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'>
```

(continues on next page)

(continued from previous page)

```
>>> 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 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,
```

(continues on next page)

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

(continued from previous page)

```
        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.

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, densifiying 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,
```

(continues on next page)

(continued from previous page)

```
                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
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 rasterio
from rasterio.warp import reproject
from rasterio.enums import Resampling

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

    # 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(
        source,
        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).

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.

```
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 excuted 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=512):
    # ...
```

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` has band objects. Rasterio does not and thus 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)
```

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 `transform` attribute of a Rasterio dataset object 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 `Afine` 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 dataset's `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 `crs` attribute of a Rasterio dataset object is an instance of Rasterio's `CRS` class and works well with `pyproj`.

```
>>> from pyproj import Proj, transform
>>> src = rasterio.open('example.tif')
>>> transform(Proj(src.crs), Proj('+init=epsg:3857'), 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 `masked_array`.

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

(continues on next page)

(continued from previous page)

```
 [[ 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')
{}
```

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 Virtual Warping

Rasterio has a `WarpedVRT` class that abstracts many of the details of raster warping by using an in-memory Warped VRT. 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
```

(continues on next page)

(continued from previous page)

```
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'

        # 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)
```

5.24.1 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

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 coordiantes 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,
```

(continues on next page)

(continued from previous page)

```
    '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.
            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')
```

5.25 Virtual Files

5.25.1 AWS S3

Note: Requires GDAL 2.1.0

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,
```

(continues on next page)

(continued from previous page)

```
#  '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 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.

5.26.1 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)
```

5.26.2 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)
```

5.26.3 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:

5.26.4 Decimation

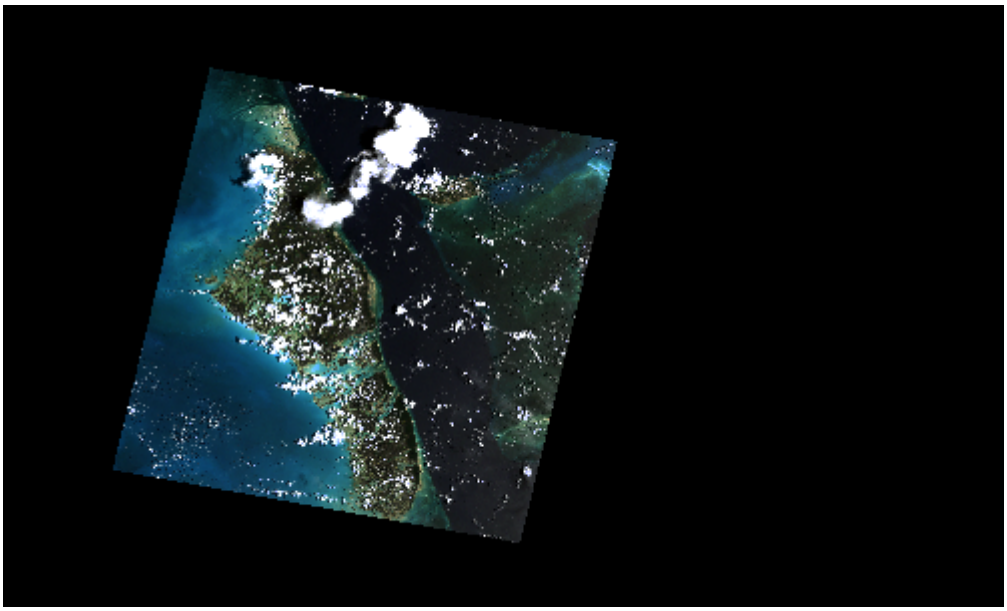
If the write window is smaller than the data, the data will be decimated. Below, the window is scaled to one third of the source image.

```
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:



5.26.5 Data windows

Sometimes it is desirable to crop off an outer boundary of NODATA values around a dataset:

```
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,
```

(continues on next page)

(continued from previous page)

```
        'transform': rasterio.windows.transform(window, src.transform)})

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

5.26.6 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)
...
>>> 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|
```

5.26.7 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)
```

5.26.8 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.

5.27 Writing Datasets

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():

    # 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.
```

(continues on next page)

(continued from previous page)

```
    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.

5.27.1 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.

CHAPTER

SIX

PYTHON API REFERENCE

6.1 rasterio package

6.1.1 Subpackages

rasterio.rio package

Submodules

rasterio.rio.blocks module

$ rio blocks

rasterio.rio.bounds module

rasterio.rio.calc module

$ rio calc

rasterio.rio.clip module

File translation command

rasterio.rio.convert module

File translation command

rasterio.rio.edit_info module

Fetch and edit raster dataset metadata from the command line.

rasterio.rio.edit_info.**all_handler**(*ctx*, *param*, *value*)
Get tags from a template file or command line.

rasterio.rio.edit_info.**colorinterp_handler**(*ctx*, *param*, *value*)
Validate a string like `red,green,blue,alpha` and convert to a tuple. Also handle `RGB` and `RGBA`.

rasterio.rio.edit_info.**crs_handler**(*ctx*, *param*, *value*)
Get crs value from a template file or command line.

rasterio.rio.edit_info.**tags_handler**(*ctx*, *param*, *value*)
Get tags from a template file or command line.

rasterio.rio.edit_info.**transform_handler**(*ctx*, *param*, *value*)
Get transform value from a template file or command line.

rasterio.rio.env module

Fetch and edit raster dataset metadata from the command line.

rasterio.rio.gcps module

Command access to dataset metadata, stats, and more.

rasterio.rio.helpers module

Helper objects used by multiple CLI commands.

rasterio.rio.helpers.**coords**(*obj*)
Yield all coordinate coordinate tuples from a geometry or feature. From python-geojson package.

rasterio.rio.helpers.**resolve_inout**(*input=None*, *output=None*, *files=None*, *overwrite=False*, *num_inputs=None*)
Resolves inputs and outputs from standard args and options.

Parameters

- **input** (*str*) – A single input filename, optional.
- **output** (*str*) – A single output filename, optional.
- **files** (*str*) – A sequence of filenames in which the last is the output filename.
- **overwrite** (*bool*) – Whether to force overwriting the output file.
- **num_inputs** (*int*) – Raise exceptions if the number of resolved input files is higher or lower than this number.

Returns

- *tuple (str, list of str)* – The resolved output filename and input filenames as a tuple of length 2.
- *If provided, the output file may be overwritten. An output*
- *file extracted from files will not be overwritten unless*

- *overwrite is True.*

Raises **click.BadParameter** –

rasterio.rio.helpers.**to_lower**(*ctx*, *param*, *value*)
Click callback, converts values to lowercase.

rasterio.rio.helpers.**write_features**(*fobj*, *collection*, *sequence=False*, *geojson_type='feature'*, *use_rs=False*, ***dump_kwds*)
Read an iterator of (feat, bbox) pairs and write to file using the selected modes.

rasterio.rio.info module

Command access to dataset metadata, stats, and more.

rasterio.rio.insp module

Fetch and edit raster dataset metadata from the command line.

class rasterio.rio.insp.**Stats**(*min*, *max*, *mean*)
Bases: tuple

property max
Alias for field number 1

property mean
Alias for field number 2

property min
Alias for field number 0

rasterio.rio.insp.**main**(*banner*, *dataset*, *alt_interpreter=None*)
Main entry point for use with python interpreter.

rasterio.rio.insp.**stats**(*dataset*)
Return a tuple with raster min, max, and mean.

rasterio.rio.main module

Main command group for Rasterio's CLI.

Subcommands developed as a part of the Rasterio package have their own modules under rasterio.rio (like rasterio/rio/info.py) and are registered in the 'rasterio.rio_commands' entry point group in Rasterio's setup.py:

entry_points="' [console_scripts] rio=rasterio.rio.main:main_group

[rasterio.rio_commands] bounds=rasterio.rio.bounds:bounds calc=rasterio.rio.calc:calc ...

Users may create their own rio subcommands by writing modules that register entry points in Rasterio's 'rasterio.rio_plugins' group. See for example https://github.com/sgillies/rio-plugin-example, which has been published to PyPI as rio-metasay.

There's no advantage to making a rio subcommand which doesn't import rasterio. But if you are using rasterio, you may profit from Rasterio's CLI infrastructure and the network of existing commands. Please add yours to the registry

https://github.com/mapbox/rasterio/wiki/Rio-plugin-registry

so that other rio users may find it.

rasterio.rio.main.**configure_logging**(*verbosity*)

rasterio.rio.main.**gdal_version_cb**(*ctx*, *param*, *value*)

rasterio.rio.mask module

rasterio.rio.merge module

$ rio merge

rasterio.rio.options module

Registry of common rio CLI options. See cligj for more options.

-a, –all: Use all pixels touched by features. In rio-mask, rio-rasterize –as-mask/–not-as-mask: interpret band as mask or not. In rio-shapes –band/–mask: use band or mask. In rio-shapes –bbox: -b, –bidx: band index(es) (singular or multiple value versions).

In rio-info, rio-sample, rio-shapes, rio-stack (different usages)

–bounds: bounds in world coordinates. In rio-info, rio-rasterize (different usages)

–count: count of bands. In rio-info –crop: Crop raster to extent of features. In rio-mask –crs: CRS of input raster. In rio-info –default-value: default for rasterized pixels. In rio-rasterize –dimensions: Output width, height. In rio-rasterize –dst-crs: destination CRS. In rio-transform –fill: fill value for pixels not covered by features. In rio-rasterize –formats: list available formats. In rio-info –height: height of raster. In rio-info -i, –invert: Invert mask created from features: In rio-mask -j, –geojson-mask: GeoJSON for masking raster. In rio-mask –lnglat: geograhpic coordinates of center of raster. In rio-info –masked/–not-masked: read masked data from source file.

In rio-calc, rio-info

-m, –mode: output file mode (r, r+). In rio-insp –name: input file name alias. In rio-calc –nodata: nodata value. In rio-info, rio-merge (different usages) –photometric: photometric interpretation. In rio-stack –property: GeoJSON property to use as values for rasterize. In rio-rasterize -r, –res: output resolution.

In rio-info, rio-rasterize (different usages. TODO: try to combine usages, prefer rio-rasterize version)

–sampling: Inverse of sampling fraction. In rio-shapes –shape: shape (width, height) of band. In rio-info –src-crs: source CRS.

In rio-insp, rio-rasterize (different usages. TODO: consolidate usages)

–stats: print raster stats. In rio-inf -t, –dtype: data type. In rio-calc, rio-info (different usages) –width: width of raster. In rio-info –with-nodata/–without-nodata: include nodata regions or not. In rio-shapes. -v, –tell-me-more, –verbose –vfs: virtual file system.

class rasterio.rio.options.**IgnoreOptionMarker**
Bases: object

A marker for an option that is to be ignored.

For use in the case where *None* is a meaningful option value, such as for nodata where *None* means “unset the nodata value.”

rasterio.rio.options.**abspath_forward_slashes**(*path*)
Return forward-slashed version of os.path.abspath

rasterio.rio.options.**bounds_handler**(*ctx*, *param*, *value*)
Handle different forms of bounds.

rasterio.rio.options.**edit_nodata_handler**(*ctx*, *param*, *value*)
Get nodata value from a template file or command line.

Expected values are 'like', 'null', a numeric value, 'nan', or IgnoreOption. Anything else should raise BadParameter.

rasterio.rio.options.**file_in_handler**(*ctx*, *param*, *value*)
Normalize ordinary filesystem and VFS paths

rasterio.rio.options.**files_in_handler**(*ctx*, *param*, *value*)
Process and validate input file names

rasterio.rio.options.**files_inout_handler**(*ctx*, *param*, *value*)
Process and validate input file names

rasterio.rio.options.**from_like_context**(*ctx*, *param*, *value*)
Return the value for an option from the context if the option or *–all* is given, else return None.

rasterio.rio.options.**like_handler**(*ctx*, *param*, *value*)
Copy a dataset's meta property to the command context for access from other callbacks.

rasterio.rio.options.**nodata_handler**(*ctx*, *param*, *value*)
Return a float or None

rasterio.rio.overview module

Manage overviews of a dataset.

rasterio.rio.overview.**build_handler**(*ctx*, *param*, *value*)

rasterio.rio.overview.**get_maximum_overview_level**(*width*, *height*, *minsize=256*)
Calculate the maximum overview level of a dataset at which the smallest overview is smaller than *minsize*.

rasterio.rio.overview.**width**
Width of the dataset.

Type int

rasterio.rio.overview.**height**
Height of the dataset.

Type int

rasterio.rio.overview.**minsize**
Minimum overview size.

Type int (default: 256)

Returns **overview_level** – overview level.

Return type int

rasterio.rio.rasterize module

$ rio rasterize

rasterio.rio.rasterize.**files_handler**(*ctx*, *param*, *value*)
Process and validate input file names

rasterio.rio.rm module

```
$ rio rm
```

rasterio.rio.sample module

rasterio.rio.shapes module

$ rio shapes

rasterio.rio.shapes.**feature_gen**(*src*, *env*, **args*, ***kwargs*)

rasterio.rio.stack module

$ rio stack

rasterio.rio.transform module

Fetch and edit raster dataset metadata from the command line.

rasterio.rio.warp module

$ rio warp

Module contents

Rasterio commandline interface components

6.1.2 Submodules

rasterio._base module

Numpy-free base classes.

class rasterio._base.**DatasetBase**
Bases: object

Dataset base class

block_shapes

bounds

closed

colorinterp

count

crs

descriptions

files

gcps

rpcs

indexes

mask_flag_enums

meta

nodata

nodatavals

profile

res

subdatasets

transform

units

compression
Compression algorithm's short name

Type str

driver
Format driver used to open the dataset

Type str

interleaving
'pixel' or 'band'

Type str

kwds
Stored creation option tags

Type dict

mode
Access mode

Type str

name
Remote or local dataset name

Type str

options
Copy of opening options

Type dict

photometric
Photometric interpretation's short name

Type str

block_shapes
An ordered list of block shapes for each bands

Shapes are tuples and have the same ordering as the dataset's shape: (count of image rows, count of image columns).

Returns

Return type list

block_size()
Returns the size in bytes of a particular block

Only useful for TIFF formatted datasets.

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type int

block_window()
Returns the window for a particular block

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type *Window*

block_windows()
Iterator over a band's blocks and their windows

The primary use of this method is to obtain windows to pass to *read()* for highly efficient access to raster block data.

The positional parameter *bidx* takes the index (starting at 1) of the desired band. This iterator yields blocks "left to right" and "top to bottom" and is similar to Python's `enumerate()` in that the first element is the block index and the second is the dataset window.

Blocks are built-in to a dataset and describe how pixels are grouped within each band and provide a mechanism for efficient I/O. A window is a range of pixels within a single band defined by row start, row stop, column start, and column stop. For example, `((0, 2), (0, 2))` defines a `2 x 2` window at the upper left corner of a raster band. Blocks are referenced by an `(i, j)` tuple where `(0, 0)` would be a band's upper left block.

Raster I/O is performed at the block level, so accessing a window spanning multiple rows in a striped raster requires reading each row. Accessing a `2 x 2` window at the center of a `1800 x 3600` image requires

reading 2 rows, or 7200 pixels just to get the target 4. The same image with internal `256 x 256` blocks would require reading at least 1 block (if the window entire window falls within a single block) and at most 4 blocks, or at least 512 pixels and at most 2048.

Given an image that is `512 x 512` with blocks that are `256 x 256`, its blocks and windows would look like:

```
Blocks:

        0       256     512
      0 +--------+--------+
        |        |        |
        | (0, 0) | (0, 1) |
        |        |        |
    256 +--------+--------+
        |        |        |
        | (1, 0) | (1, 1) |
        |        |        |
    512 +--------+--------+


Windows:

    UL: ((0, 256), (0, 256))
    UR: ((0, 256), (256, 512))
    LL: ((256, 512), (0, 256))
    LR: ((256, 512), (256, 512))
```

Parameters **bidx** (*int, optional*) – The band index (using 1-based indexing) from which to extract windows. A value less than 1 uses the first band if all bands have homogeneous windows and raises an exception otherwise.

Yields *block, window*

bounds

Returns the lower left and upper right bounds of the dataset in the units of its coordinate reference system.

The returned value is a tuple: (lower left x, lower left y, upper right x, upper right y)

checksum()

Compute an integer checksum for the stored band

Parameters

- **bidx** (*int*) – The band's index (1-indexed).
- **window** (*tuple, optional*) – A window of the band. Default is the entire extent of the band.

Returns

Return type An int.

close()

Close the dataset

closed

Test if the dataset is closed

Returns

Return type bool

colorinterp
Returns a sequence of `ColorInterp.<enum>` representing color interpretation in band order.

To set color interpretation, provide a sequence of `ColorInterp.<enum>`:

import rasterio from rasterio.enums import ColorInterp

with rasterio.open('rgba.tif', 'r+') as src:

src.colorinterp = (ColorInterp.red, ColorInterp.green, ColorInterp.blue, ColorInterp.alpha)

Returns

Return type tuple

colormap()
Returns a dict containing the colormap for a band or None.

compression

count
The number of raster bands in the dataset

Returns

Return type int

crs
The dataset's coordinate reference system

In setting this property, the value may be a CRS object or an EPSG:nnnn or WKT string.

Returns

Return type *CRS*

descriptions
Descriptions for each dataset band

To set descriptions, one for each band is required.

Returns

Return type list of str

driver

dtypes
The data types of each band in index order

Returns

Return type list of str

files
Returns a sequence of files associated with the dataset.

Returns

Return type tuple

gcps
ground control points and their coordinate reference system.

This property is a 2-tuple, or pair: (gcps, crs).

gcps [list of GroundControlPoint] Zero or more ground control points.

crs: CRS The coordinate reference system of the ground control points.

get_gcps()
Get GCPs and their associated CRS.

get_nodatavals()

get_tag_item()
Returns tag item value

Parameters

- **ns** (*str*) – The key for the metadata item to fetch.
- **dm** (*str*) – The domain to fetch for.
- **bidx** (*int*) – Band index, starting with 1.
- **ovr** (*int*) – Overview level

Returns

Return type str

get_transform()
Returns a GDAL geotransform in its native form.

height

indexes
The 1-based indexes of each band in the dataset

For a 3-band dataset, this property will be `[1, 2, 3]`.

Returns

Return type list of int

interleaving

is_tiled

lnglat()

mask_flag_enums
Sets of flags describing the sources of band masks.

all_valid: There are no invalid pixels, all mask values will be

255. When used this will normally be the only flag set.

per_dataset: The mask band is shared between all bands on the dataset.

alpha: The mask band is actually an alpha band and may have values other than 0 and 255.

nodata: Indicates the mask is actually being generated from nodata values (mutually exclusive of "alpha").

Returns One list of rasterio.enums.MaskFlags members per band.

Return type list [, list*]

Examples

For a 3 band dataset that has masks derived from nodata values:

```
>>> dataset.mask_flag_enums
([<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>])
>>> band1_flags = dataset.mask_flag_enums[0]
>>> rasterio.enums.MaskFlags.nodata in band1_flags
True
>>> rasterio.enums.MaskFlags.alpha in band1_flags
False
```

meta
The basic metadata of this dataset.

mode

name

nodata
The dataset's single nodata value

Notes

May be set.

Returns

Return type float

nodatavals
Nodata values for each band

Notes

This may not be set.

Returns

Return type list of float

offsets
Raster offset for each dataset band

To set offsets, one for each band is required.

Returns

Return type list of float

options

overviews()

photometric

profile
Basic metadata and creation options of this dataset.

May be passed as keyword arguments to *rasterio.open()* to create a clone of this dataset.

read_crs()
Return the GDAL dataset's stored CRS

read_transform()
Return the stored GDAL GeoTransform

res
Returns the (width, height) of pixels in the units of its coordinate reference system.

rpcs
Rational polynomial coefficients mapping between pixel and geodetic coordinates.

This property is a dict-like object.

rpcs : RPC instance containing coefficients. Empty if dataset does not have any metadata in the "RPC" domain.

scales
Raster scale for each dataset band

To set scales, one for each band is required.

Returns

Return type list of float

shape

start()
Start the dataset's life cycle

stop()
Close the GDAL dataset handle

subdatasets
Sequence of subdatasets

tag_namespaces()
Get a list of the dataset's metadata domains.

Returned items may be passed as *ns* to the tags method.

Parameters

- **int** (*bidx*) – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.
- **optional** – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.

Returns

Return type list of str

tags()
Returns a dict containing copies of the dataset or band's tags.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the tags of a specific band. The optional ns argument can be used to select a namespace other than the default.

transform
The dataset's georeferencing transformation matrix

This transform maps pixel row/column coordinates to coordinates in the dataset's coordinate reference system.

Returns

Return type Affine

units
one units string for each dataset band

Possible values include 'meters' or 'degC'. See the Pint project for a suggested list of units.

To set units, one for each band is required.

Returns

Return type list of str

Type A list of str

width

write_transform()

rasterio._base.**check_gdal_version**()
Return True if the major and minor versions match.

rasterio._base.**driver_can_create**()
Return True if the driver has CREATE capability

rasterio._base.**driver_can_create_copy**()
Return True if the driver has CREATE_COPY capability

rasterio._base.**driver_supports_mode**()
Return True if the driver supports the mode

rasterio._base.**gdal_version**()
Return the version as a major.minor.patchlevel string.

rasterio._base.**get_dataset_driver**()
Return the name of the driver that opens a dataset

Parameters **path** (*rasterio.path.Path or str*) – A remote or local dataset path.

Returns

Return type str

rasterio._crs module

Coordinate reference systems, class and functions.

rasterio._crs.**gdal_version**()
Return the version as a major.minor.patchlevel string.

rasterio._env module

GDAL and OGR driver and configuration management

The main thread always utilizes CPLSetConfigOption. Child threads utilize CPLSetThreadLocalConfigOption instead. All threads use CPLGetConfigOption and not CPLGetThreadLocalConfigOption, thus child threads will inherit config options from the main thread unless the option is set to a new value inside the thread.

class rasterio._env.**ConfigEnv**
Bases: object

Configuration option management

clear_config_options()
Clear GDAL config options.

get_config_options()

options

update_config_options()
Update GDAL config options.

class rasterio._env.**GDALDataFinder**
Bases: object

Finds GDAL data files

Note: this is not part of the public API in 1.0.x.

find_file(*basename*)
Returns path of a GDAL data file or None

Parameters **basename** (*str*) – Basename of a data file such as "header.dxf"

Returns

Return type str (on success) or None (on failure)

search(*prefix=None*)
Returns GDAL data directory

Note well that os.environ is not consulted.

Returns

Return type str or None

search_debian(*prefix='/home/docs/checkouts/readthedocs.org/user_builds/rasterio-spestana/conda/latest'*)
Check Debian locations

search_prefix(*prefix='/home/docs/checkouts/readthedocs.org/user_builds/rasterio-spestana/conda/latest'*)
Check sys.prefix location

search_wheel(*prefix=None*)
Check wheel location

class rasterio._env.**GDALEnv**
Bases: *rasterio._env.ConfigEnv*

Configuration and driver management

drivers()

start()

stop()

class rasterio._env.**PROJDataFinder**
Bases: object

Finds PROJ data files

Note: this is not part of the public API in 1.0.x.

has_data()
Returns True if PROJ's data files can be found

Returns

Return type bool

search(*prefix=None*)
Returns PROJ data directory

Note well that os.environ is not consulted.

Returns

Return type str or None

search_prefix(*prefix='/home/docs/checkouts/readthedocs.org/user_builds/rasterio-spestana/conda/latest'*)
Check sys.prefix location

search_wheel(*prefix=None*)
Check wheel location

rasterio._env.**catch_errors**()
Intercept GDAL errors

rasterio._env.**del_gdal_config**()
Delete a GDAL configuration option.

Parameters **key** (*str*) – Name of config option.

rasterio._env.**driver_count**()
Return the count of all drivers

rasterio._env.**get_gdal_config**()
Get the value of a GDAL configuration option. When requesting GDAL_CACHEMAX the value is returned unaltered.

Parameters

- **key** (*str*) – Name of config option.
- **normalize** (*bool, optional*) – Convert values of "ON"' and "OFF" to True and False.

rasterio._env.**set_gdal_config**()
Set a GDAL configuration option's value.

Parameters

- **key** (*str*) – Name of config option.
- **normalize** (*bool, optional*) – Convert True to *"ON"* and False to *"OFF"*`.

rasterio._env.**set_proj_data_search_path**()
Set PROJ data search path

rasterio._err module

rasterio._err

Exception-raising wrappers for GDAL API functions.

exception rasterio._err.**CPLE_AWSAccessDeniedError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AWSBucketNotFoundError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AWSError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AWSInvalidCredentialsError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AWSObjectNotFoundError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AWSSignatureDoesNotMatchError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AppDefinedError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_AssertionFailedError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_BaseError**(*error*, *errno*, *errmsg*)
Bases: Exception

Base CPL error class

Exceptions deriving from this class are intended for use only in Rasterio's Cython code. Let's not expose API users to them.

property args

exception rasterio._err.**CPLE_FileIOError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_HttpResponseError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_IllegalArgError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_NoWriteAccessError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_NotSupportedError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_OpenFailedError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_OutOfMemoryError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

exception rasterio._err.**CPLE_UserInterruptError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

class rasterio._err.**GDALError**(*value*)
Bases: enum.IntEnum

An enumeration.

debug = 1

failure = 3

fatal = 4

none = 0

warning = 2

exception rasterio._err.**ObjectNullError**(*error*, *errno*, *errmsg*)
Bases: *rasterio._err.CPLE_BaseError*

rasterio._example module

rasterio._example.**compute**()
reverses bands inefficiently

Given input and output uint8 arrays, fakes an CPU-intensive computation.

rasterio._features module

Feature extraction

class rasterio._features.**GeomBuilder**
Bases: object

Builds a GeoJSON (Fiona-style) geometry from an OGR geometry.

class rasterio._features.**OGRGeomBuilder**
Bases: object

Builds an OGR geometry from GeoJSON geometry. From Fiona: https://github.com/Toblerity/Fiona/blob/master/src/fiona/ogrext.pyx

class rasterio._features.**ShapeIterator**
Bases: object

Provides an iterator over shapes in an OGR feature layer.

rasterio._fill module

Raster fill.

rasterio._io module

Rasterio input/output.

class rasterio._io.**BufferedDatasetWriterBase**
Bases: *rasterio._io.DatasetWriterBase*

stop()
Close the GDAL dataset handle

class rasterio._io.**DatasetReaderBase**
Bases: *rasterio._base.DatasetBase*

dataset_mask()
Get the dataset's 2D valid data mask.

Parameters

- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot be combined with *out_shape*.

- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot be combined with *out*.

- **window**(*a pair (tuple) of pairs of ints or Window, optional*)– The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns The dtype of this array is uint8. 0 = nodata, 255 = valid data.

Return type Numpy ndarray or a view on a Numpy ndarray

Notes

Note: as with Numpy ufuncs, an object is returned even if you use the optional *out* argument and the return value shall be preferentially used by callers.

The dataset mask is calculated based on the individual band masks according to the following logic, in order of precedence:

1. If a .msk file, dataset-wide alpha, or internal mask exists it will be used for the dataset mask.
2. Else if the dataset is a 4-band 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 255.

Note that this differs from read_masks and GDAL RFC15 in that it applies per-dataset, not per-band (see https://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask)

read()

Read a dataset's raw pixels as an N-d array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array into which data will be placed. If the height and width of *out* differ from that of the specified window (see below), the raster image will be decimated or replicated using the specified resampling method (also see below).

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  This parameter cannot be combined with *out_shape*.
- **out_dtype** (*str or numpy dtype*) – The desired output data type. For example: 'uint8' or rasterio.uint16.
- **out_shape** (*tuple, optional*) – A tuple describing the shape of a new output array. See *out* (above) for notes on image decimation and replication.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **masked** (*bool, optional*) – If *masked* is *True* the return value will be a masked array. Otherwise (the default) the return value will be a regular array. Masks will be exactly the inverse of the GDAL RFC 15 conforming arrays returned by read_masks().
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.
- **fill_value** (*scalar*) – Fill value applied in the *boundless=True* case only. Like the fill_value of numpy.ma.MaskedArray, should be value valid for the dataset's data type.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be

- *preferentially used by callers.*

read_masks()
Read raster band masks as a multidimensional array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot combine with *out_shape*.
- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

sample()
Get the values of a dataset at certain positions

Values are from the nearest pixel. They are not interpolated.

Parameters

- **xy** (*iterable*) – Pairs of x, y coordinates (floats) in the dataset's reference system.
- **indexes** (*int or list of int*) – Indexes of dataset bands to sample.
- **masked** (*bool, default: False*) – Whether to mask samples that fall outside the extent of the dataset.

Returns Arrays of length equal to the number of specified indexes containing the dataset values for the bands corresponding to those indexes.

Return type iterable

class rasterio._io.**DatasetWriterBase**

Bases: *rasterio._io.DatasetReaderBase*

Read-write access to raster data and metadata

build_overviews()

Build overviews at one or more decimation factors for all bands of the dataset.

set_band_description()

Sets the description of a dataset band.

Parameters

- **bidx** (*int*) – Index of the band (starting with 1).
- **value** (*string*) – A description of the band.

Returns

Return type None

set_band_unit()

Sets the unit of measure of a dataset band.

Parameters

- **bidx** (*int*) – Index of the band (starting with 1).
- **value** (*string*) – A label for the band's unit of measure such as 'meters' or 'degC'. See the Pint project for a suggested list of units.

Returns

Return type None

update_tags()

Updates the tags of a dataset or one of its bands.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the dataset band. The optional ns argument can be used to select a namespace other than the default.

write()

Write the arr array into indexed bands of the dataset.

If *indexes* is a list, the src must be a 3D array of matching shape. If an int, the src must be a 2D array.

See *read()* for usage of the optional *window* argument.

write_band()

Write the src array into the *bidx* band.

Band indexes begin with 1: read_band(1) returns the first band.

The optional *window* argument takes a tuple like:

((row_start, row_stop), (col_start, col_stop))

specifying a raster subset to write into.

write_colormap()
Write a colormap for a band to the dataset.

write_mask()
Write the valid data mask src array into the dataset's band mask.

The optional *window* argument takes a tuple like:

((row_start, row_stop), (col_start, col_stop))

specifying a raster subset to write into.

write_transform()

class rasterio._io.**InMemoryRaster**
Bases: object

Class that manages a single-band in memory GDAL raster dataset. Data type is determined from the data type of the input numpy 2D array (image), and must be one of the data types supported by GDAL (see rasterio.dtypes.dtype_rev). Data are populated at create time from the 2D array passed in.

Use the 'with' pattern to instantiate this class for automatic closing of the memory dataset.

This class includes attributes that are intended to be passed into GDAL functions: self.dataset self.band self.band_ids (single element array with band ID of this dataset's band) self.transform (GDAL compatible transform array)

This class is only intended for internal use within rasterio to support IO with GDAL. Other memory based operations should use numpy arrays.

close()

read()

write()

class rasterio._io.**MemoryFileBase**
Bases: object

Base for a BytesIO-like class backed by an in-memory file.

close()

exists()
Test if the in-memory file exists.

Returns True if the in-memory file exists.

Return type bool

getbuffer()
Return a view on bytes of the file.

read()
Read size bytes from MemoryFile.

seek()

tell()

write()
Write data bytes to MemoryFile

rasterio._io.**silence_errors**()
Intercept GDAL errors

rasterio._io.**validate_resampling**()
Validate that the resampling method is compatible of reads/writes

rasterio._io.**virtual_file_to_buffer**()
Read content of a virtual file into a Python bytes buffer.

rasterio._warp module

Raster and vector warping and reprojection.

class rasterio._warp.**WarpedVRTReaderBase**
Bases: *rasterio._io.DatasetReaderBase*

crs
The dataset's coordinate reference system

read()
Read a dataset's raw pixels as an N-d array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array into which data will be placed. If the height and width of *out* differ from that of the specified window (see below), the raster image will be decimated or replicated using the specified resampling method (also see below).

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  This parameter cannot be combined with *out_shape*.
- **out_dtype** (*str or numpy dtype*) – The desired output data type. For example: 'uint8' or rasterio.uint16.
- **out_shape** (*tuple, optional*) – A tuple describing the shape of a new output array. See *out* (above) for notes on image decimation and replication.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **masked** (*bool, optional*) – If *masked* is *True* the return value will be a masked array. Otherwise (the default) the return value will be a regular array. Masks will be exactly the inverse of the GDAL RFC 15 conforming arrays returned by read_masks().
- **resampling** (*Resampling*) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.
- **fill_value** (*scalar*) – Fill value applied in the *boundless=True* case only.

- **kwargs** (*dict*) – This is only for backwards compatibility. No keyword arguments are supported other than the ones named above.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_masks()
Read raster band masks as a multidimensional array

rasterio._warp.**recursive_round**()
Recursively round coordinates.

rasterio.compat module

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

Bounding box tuple, and disjoint operator.

class rasterio.coords.**BoundingBox**(*left*, *bottom*, *right*, *top*)
Bases: *rasterio.coords.BoundingBox*

Bounding box named tuple, defining extent in cartesian coordinates.

```
BoundingBox(left, bottom, right, top)
```

left
Left coordinate

bottom
Bottom coordinate

right
Right coordinate

top
Top coordinate

rasterio.coords.**disjoint_bounds**(*bounds1*, *bounds2*)
Compare two bounds and determine if they are disjoint.

Parameters

- **bounds1** (*4-tuple*) – rasterio bounds tuple (left, bottom, right, top)
- **bounds2** (*4-tuple*) – rasterio bounds tuple

Returns

- *boolean*
- `True` if bounds are disjoint,
- `False` if bounds overlap

rasterio.crs module

Coordinate Reference Systems

Notes

In Rasterio versions <= 1.0.13, coordinate reference system support was limited to the CRS that can be described by PROJ parameters. This limitation is gone in versions >= 1.0.14. Any CRS that can be defined using WKT (version 1) may be used.

class `rasterio.crs.`**CRS**(*initialdata=None*, ***kwargs*)

Bases: `collections.abc.Mapping`

A geographic or projected coordinate reference system

CRS objects may be created by passing PROJ parameters as keyword arguments to the standard constructor or by passing EPSG codes, PROJ mappings, PROJ strings, or WKT strings to the from_epsg, from_dict, from_string, or from_wkt class methods or static methods.

Examples

The from_dict method takes PROJ parameters as keyword arguments.

```
>>> crs = CRS.from_dict(init='epsg:3005')
```

EPSG codes may be used with the from_epsg method.

```
>>> crs = CRS.from_epsg(3005)
```

The from_string method takes a variety of input.

```
>>> crs = CRS.from_string('EPSG:3005')
```

property data

A PROJ4 dict representation of the CRS

classmethod from_authority(*auth_name*, *code*)

Make a CRS from an authority name and authority code

New in version 1.1.7.

Parameters

- **auth_name** (*str*) – The name of the authority.
- **code** (*int or str*) – The code used by the authority.

Returns

Return type *CRS*

classmethod from_dict(*initialdata=None*, ***kwargs*)
Make a CRS from a PROJ dict

Parameters

- **initialdata** (*mapping, optional*) – A dictionary or other mapping
- **kwargs** (*mapping, optional*) – Another mapping. Will be overlaid on the initialdata.

Returns

Return type *CRS*

classmethod from_epsg(*code*)
Make a CRS from an EPSG code

Parameters **code** (*int or str*) – An EPSG code. Strings will be converted to integers.

Notes

The input code is not validated against an EPSG database.

Returns

Return type *CRS*

classmethod from_proj4(*proj*)
Make a CRS from a PROJ4 string

Parameters **proj** (*str*) – A PROJ4 string like "+proj=longlat ..."

Returns

Return type *CRS*

classmethod from_string(*string*, *morph_from_esri_dialect=False*)
Make a CRS from an EPSG, PROJ, or WKT string

Parameters

- **string** (*str*) – An EPSG, PROJ, or WKT string.
- **morph_from_esri_dialect** (*bool, optional*) – If True, items in the input using Esri's dialect of WKT will be replaced by OGC standard equivalents.

Returns

Return type *CRS*

classmethod from_user_input(*value*, *morph_from_esri_dialect=False*)
Make a CRS from various input

Dispatches to from_epsg, from_proj, or from_string

Parameters

- **value** (*obj*) – A Python int, dict, or str.
- **morph_from_esri_dialect** (*bool, optional*) – If True, items in the input using Esri's dialect of WKT will be replaced by OGC standard equivalents.

Returns

Return type *CRS*

classmethod from_wkt(*wkt*, *morph_from_esri_dialect=False*)
Make a CRS from a WKT string

Parameters

- **wkt** (*str*) – A WKT string.
- **morph_from_esri_dialect** (*bool, optional*) – If True, items in the input using Esri's dialect of WKT will be replaced by OGC standard equivalents.

Returns

Return type *CRS*

property is_epsg_code
Test if the CRS is defined by an EPSG code

Returns

Return type bool

property is_geographic
Test that the CRS is a geographic CRS

Returns

Return type bool

property is_projected
Test that the CRS is a projected CRS

Returns

Return type bool

property is_valid
Test that the CRS is a geographic or projected CRS

Notes

There are other types of CRS, such as compound or local or engineering CRS, but these are not supported in Rasterio 1.0.

Returns

Return type bool

property linear_units
The linear units of the CRS

Possible values include "metre" and "US survey foot".

Returns

Return type str

property linear_units_factor
The linear units of the CRS and the conversion factor to meters.

The first element of the tuple is a string, its possible values include "metre" and "US survey foot". The second element of the tuple is a float that represent the conversion factor of the raster units to meters.

Returns

Return type tuple

to_authority()
The authority name and code of the CRS

New in version 1.1.7.

Returns

Return type (str, str) or None

to_dict()
Convert CRS to a PROJ4 dict

Notes

If there is a corresponding EPSG code, it will be used.

Returns

Return type dict

to_epsg()
The epsg code of the CRS

Returns None if there is no corresponding EPSG code.

Returns

Return type int

to_proj4()
Convert CRS to a PROJ4 string

Returns

Return type str

to_string()
Convert CRS to a PROJ4 or WKT string

Notes

Mapping keys are tested against the `all_proj_keys` list. Values of `True` are omitted, leaving the key bare: {'no_defs': True} -> "+no_defs" and items where the value is otherwise not a str, int, or float are omitted.

Returns

Return type str

to_wkt(*morph_to_esri_dialect=False*, *version=None*)
Convert CRS to its OGC WKT representation

New in version 1.3.0: version

Parameters

- **morph_to_esri_dialect** (*bool, optional*) – Whether or not to morph to the Esri dialect of WKT. Only works for GDAL 2.
- **version** (*WktVersion or str, optional*) – The version of the WKT output. Only works with GDAL 3+. Default is WKT1_GDAL.

Returns

Return type str

property wkt
An OGC WKT representation of the CRS

Returns

Return type str

rasterio.crs.**epsg_treats_as_latlong**(*input*)
Test if the CRS is in latlon order

From GDAL docs:

> This method returns TRUE if EPSG feels this geographic coordinate system should be treated as having lat/long coordinate ordering.

> Currently this returns TRUE for all geographic coordinate systems with an EPSG code set, and axes set defining it as lat, long.

> FALSE will be returned for all coordinate systems that are not geographic, or that do not have an EPSG code set.

> **Note**

> Important change of behavior since GDAL 3.0. In previous versions, geographic CRS imported with importFromEPSG() would cause this method to return FALSE on them, whereas now it returns TRUE, since importFromEPSG() is now equivalent to importFromEPSGA().

Parameters **input** (CRS) – Coordinate reference system, as a rasterio CRS object Example: CRS({'init': 'EPSG:4326'})

Returns

Return type bool

rasterio.crs.**epsg_treats_as_northingeasting**(*input*)
Test if the CRS should be treated as having northing/easting coordinate ordering

From GDAL docs:

> This method returns TRUE if EPSG feels this projected coordinate system should be treated as having northing/easting coordinate ordering.

> Currently this returns TRUE for all projected coordinate systems with an EPSG code set, and axes set defining it as northing, easting.

> FALSE will be returned for all coordinate systems that are not projected, or that do not have an EPSG code set.

> **Note**

> Important change of behavior since GDAL 3.0. In previous versions, projected CRS with northing, easting axis order imported with importFromEPSG() would cause this method to return FALSE on them, whereas now it returns TRUE, since importFromEPSG() is now equivalent to importFromEPSGA().

Parameters **input** (CRS) – Coordinate reference system, as a rasterio CRS object Example: CRS({'init': 'EPSG:4326'})

Returns

Return type bool

rasterio.drivers module

Driver policies and utilities

GDAL has many standard and extension format drivers and completeness of these drivers varies greatly. It's possible to succeed poorly with some formats and drivers, meaning that easy problems can be solved but that harder problems are blocked by limitations of the drivers and formats.

NetCDF writing, for example, is presently blacklisted. Rasterio users should use netcdf4-python instead: http://unidata.github.io/netcdf4-python/.

`rasterio.drivers.`**`driver_from_extension`**(*path*)
Attempt to auto-detect driver based on the extension.

Parameters **`path`** (*`str or pathlike object`*) – The path to the dataset to write with.

Returns The name of the driver for the extension.

Return type str

`rasterio.drivers.`**`is_blacklisted`**(*name*, *mode*)
Returns True if driver *name* and *mode* are blacklisted.

`rasterio.drivers.`**`raster_driver_extensions`**()

Returns Map of extensions to the driver.

Return type dict

rasterio.dtypes module

Mapping of GDAL to Numpy data types.

Since 0.13 we are not importing numpy here and data types are strings. Happily strings can be used throughout Numpy and so existing code will not break.

Within Rasterio, to test data types, we use Numpy's dtype() factory to do something like this:

if np.dtype(destination.dtype) == np.dtype(rasterio.uint8): ...

`rasterio.dtypes.`**`can_cast_dtype`**(*values*, *dtype*)
Test if values can be cast to dtype without loss of information.

Parameters

- **`values`** (*`list-like`*) –
- **`dtype`** (*`numpy dtype or string`*) –

Returns True if values can be cast to data type.

Return type boolean

`rasterio.dtypes.`**`check_dtype`**(*dt*)
Check if dtype is a known dtype.

`rasterio.dtypes.`**`get_minimum_dtype`**(*values*)
Determine minimum type to represent values.

Uses range checking to determine the minimum integer or floating point data type required to represent values.

Parameters **`values`** (*`list-like`*) –

Returns

Return type rasterio dtype string

rasterio.dtypes.**is_ndarray**(*array*)
Check if array is a ndarray.

rasterio.dtypes.**validate_dtype**(*values*, *valid_dtypes*)
Test if dtype of values is one of valid_dtypes.

Parameters

- **values** (*list-like*) –
- **valid_dtypes** (*list-like*) – list of valid dtype strings, e.g., ('int16', 'int32')

Returns True if dtype of values is one of valid_dtypes

Return type boolean

rasterio.enums module

Enumerations.

class rasterio.enums.**ColorInterp**(*value*)
Bases: enum.IntEnum

Raster band color interpretation.

Cb = 15

Cr = 16

Y = 14

alpha = 6

black = 13

blue = 5

cyan = 10

gray = 1

green = 4

grey = 1

hue = 7

lightness = 9

magenta = 11

palette = 2

red = 3

saturation = 8

undefined = 0

yellow = 12

class rasterio.enums.**Compression**(*value*)
Bases: enum.Enum

Available compression algorithms.

ccittfax3 = 'CCITTFAX3'

ccittfax4 = 'CCITTFAX4'

ccittrle = 'CCITTRLE'

deflate = 'DEFLATE'

jpeg = 'JPEG'

jpeg2000 = 'JPEG2000'

lerc = 'LERC'

lzma = 'LZMA'

lzw = 'LZW'

none = 'NONE'

packbits = 'PACKBITS'

webp = 'WEBP'

zstd = 'ZSTD'

class rasterio.enums.**Interleaving**(*value*)
Bases: enum.Enum

An enumeration.

band = 'BAND'

line = 'LINE'

pixel = 'PIXEL'

class rasterio.enums.**MaskFlags**(*value*)
Bases: enum.IntEnum

An enumeration.

all_valid = 1

alpha = 4

nodata = 8

per_dataset = 2

class rasterio.enums.**MergeAlg**(*value*)
Bases: enum.Enum

Available rasterization algorithms

add = 'ADD'

replace = 'REPLACE'

class rasterio.enums.**PhotometricInterp**(*value*)
Bases: enum.Enum

An enumeration.

black = 'MINISBLACK'

cielab = 'CIELAB'

cmyk = 'CMYK'

icclab = 'ICCLAB'

itulab = 'ITULAB'

rgb = 'RGB'

white = 'MINISWHITE'

ycbcr = 'YCbCr'

class rasterio.enums.**Resampling**(*value*)
Bases: enum.IntEnum

Available warp resampling algorithms.

nearest
Nearest neighbor resampling (default, fastest algorithm, worst interpolation quality).

bilinear
Bilinear resampling.

cubic
Cubic resampling.

cubic_spline
Cubic spline resampling.

lanczos
Lanczos windowed sinc resampling.

average
Average resampling, computes the weighted average of all non-NODATA contributing pixels.

mode
Mode resampling, selects the value which appears most often of all the sampled points.

gauss
Gaussian resampling, Note: not available to the functions in rio.warp.

max
Maximum resampling, selects the maximum value from all non-NODATA contributing pixels. (GDAL >= 2.0)

min
Minimum resampling, selects the minimum value from all non-NODATA contributing pixels. (GDAL >= 2.0)

med
Median resampling, selects the median value of all non-NODATA contributing pixels. (GDAL >= 2.0)

q1
Q1, first quartile resampling, selects the first quartile value of all non-NODATA contributing pixels. (GDAL >= 2.0)

q3
Q3, third quartile resampling, selects the third quartile value of all non-NODATA contributing pixels. (GDAL >= 2.0)

sum
Sum, compute the weighted sum of all non-NODATA contributing pixels. (GDAL >= 3.1)

rms
RMS, root mean square / quadratic mean of all non-NODATA contributing pixels. (GDAL >= 3.3)

Notes

The first 8, 'nearest', 'bilinear', 'cubic', 'cubic_spline', 'lanczos', 'average', 'mode', and 'gauss', are available for making dataset overviews.

'max', 'min', 'med', 'q1', 'q3' are only supported in GDAL >= 2.0.0.

'nearest', 'bilinear', 'cubic', 'cubic_spline', 'lanczos', 'average', 'mode' are always available (GDAL >= 1.10).

'sum' is only supported in GDAL >= 3.1.

'rms' is only supported in GDAL >= 3.3.

Note: 'gauss' is not available to the functions in rio.warp.

average = 5

bilinear = 1

cubic = 2

cubic_spline = 3

gauss = 7

lanczos = 4

max = 8

med = 10

min = 9

mode = 6

nearest = 0

q1 = 11

q3 = 12

rms = 14

sum = 13

class rasterio.enums.**WktVersion**(*value*)

Bases: enum.Enum

New in version 1.3.0.

Supported CRS WKT string versions

WKT1 = 'WKT1'
Alias for WKT Version 1 GDAL Style

WKT1_ESRI = 'WKT1_ESRI'
WKT Version 1 ESRI Style

WKT1_GDAL = 'WKT1_GDAL'
WKT Version 1 GDAL Style

WKT2 = 'WKT2'
Alias for latest WKT Version 2

WKT2_2015 = 'WKT2_2015'
WKT Version 2 from 2015

WKT2_2019 = 'WKT2_2018'
WKT Version 2 from 2019

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.

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.

Returns

Return type None

classmethod default_options()
Default configuration options

Parameters **None** –

Returns

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()

Returns

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*) –

Returns

Return type GDALVersion instance

classmethod runtime()
Return GDALVersion of current GDAL runtime

class rasterio.env.**NullContextManager**
Bases: object

class rasterio.env.**ThreadEnv**
Bases: _thread._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.

Returns

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

Returns

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 GDALVersionErorr.

@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.

Returns

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: *rasterio.errors.RasterioError*, NotImplementedError

Raised when dataset attributes are misused

exception rasterio.errors.**DriverCapabilityError**
Bases: *rasterio.errors.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: *rasterio.errors.RasterioError*

Raised when the state of GDAL/AWS environment cannot be created or modified.

exception rasterio.errors.**FileOverwriteError**(*message*)
Bases: click.exceptions.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.

Raises an exception on GDAL >= 2.2.0 rasterio.warp.transform_geometry(

src_crs, dst_crs, antimeridian_cutting=False)

exception rasterio.errors.**GDALOptionNotImplementedError**
Bases: *rasterio.errors.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: *rasterio.errors.RasterioError*

Raised if the runtime version of GDAL does not meet the required version of GDAL.

exception rasterio.errors.**InvalidArrayError**
Bases: *rasterio.errors.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: *rasterio.errors.RasterioError*

Raised when creation of an overview fails

exception rasterio.errors.**PathError**
Bases: *rasterio.errors.RasterioError*

Raised when a dataset path is malformed or invalid

exception rasterio.errors.**RPCTransformWarning**
Bases: UserWarning

Error raised when GDALRPCTransform fails.

exception rasterio.errors.**RasterBlockError**
Bases: *rasterio.errors.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: *rasterio.errors.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.**TransformError**
Bases: *rasterio.errors.RasterioError*

Raised when transform arguments are invalid

exception rasterio.errors.**UnsupportedOperation**
Bases: *rasterio.errors.RasterioError*

Raised when reading from a file opened in 'w' mode

exception rasterio.errors.**WarpOptionsError**
Bases: *rasterio.errors.RasterioError*

Raised when options for a warp operation are invalid

exception rasterio.errors.**WarpedVRTError**
Bases: *rasterio.errors.RasterioError*

Raised when WarpedVRT can't be initialized

exception rasterio.errors.**WindowError**
Bases: *rasterio.errors.RasterioError*

Raised when errors occur during window operations

exception rasterio.errors.**WindowEvaluationError**
Bases: ValueError

Raised when window evaluation fails

rasterio.features module

Functions for working with features in a raster dataset.

rasterio.features.**bounds**(*geometry*, *north_up=True*, *transform=None*)
Return a (left, bottom, right, top) bounding box.

From Fiona 1.4.8. Modified to return bbox from geometry if available.

Parameters **geometry** (*GeoJSON-like feature (implements __geo_interface__),*) – feature collection, or geometry.

Returns Bounding box: (left, bottom, right, top)

Return type tuple

rasterio.features.**dataset_features**(*src*, *bidx=None*, *sampling=1*, *band=True*, *as_mask=False*, *with_nodata=False*, *geographic=True*, *precision=- 1*)
Yield GeoJSON features for the dataset

The geometries are polygons bounding contiguous regions of the same raster value.

Parameters

- **src** (*Rasterio Dataset*) –
- **bidx** (*int*) – band index
- **sampling** (*int (DEFAULT: 1)*) – Inverse of the sampling fraction; a value of 10 decimates
- **band** (*boolean (DEFAULT: True)*) – extract features from a band (True) or a mask (False)
- **as_mask** (*boolean (DEFAULT: False)*) – Interpret band as a mask and output only one class of valid data shapes?
- **with_nodata** (*boolean (DEFAULT: False)*) – Include nodata regions?

- **geographic** (*str (DEFAULT: True)*) – Output shapes in EPSG:4326? Otherwise use the native CRS.
- **precision** (*int (DEFAULT: -1)*) – Decimal precision of coordinates. -1 for full float precision output

Yields *GeoJSON-like Feature dictionaries for shapes found in the given band*

rasterio.features.**geometry_mask**(*geometries*, *out_shape*, *transform*, *all_touched=False*, *invert=False*)

Create a mask from shapes.

By default, mask is intended for use as a numpy mask, where pixels that overlap shapes are False.

Parameters

- **geometries** (*iterable over geometries (GeoJSON-like objects)*) –
- **out_shape** (*tuple or list*) – Shape of output numpy ndarray.
- **transform** (*Affine transformation object*) – Transformation from pixel coordinates of *source* to the coordinate system of the input *shapes*. See the *transform* property of dataset objects.
- **all_touched** (*boolean, optional*) – If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in.
- **invert** (*boolean, optional*) – If True, mask will be True for pixels that overlap shapes. False by default.

Returns Result

Return type numpy ndarray of type 'bool'

Notes

See rasterize() for performance notes.

rasterio.features.**geometry_window**(*dataset*, *shapes*, *pad_x=0*, *pad_y=0*, *north_up=None*, *rotated=None*, *pixel_precision=None*, *boundless=False*)

Calculate the window within the raster that fits the bounds of the geometry plus optional padding. The window is the outermost pixel indices that contain the geometry (floor of offsets, ceiling of width and height).

If shapes do not overlap raster, a WindowError is raised.

Parameters

- **dataset** (*dataset object opened in 'r' mode*) – Raster for which the mask will be created.
- **shapes** (*iterable over geometries.*) – A geometry is a GeoJSON-like object or implements the geo interface. Must be in same coordinate system as dataset.
- **pad_x** (*float*) – Amount of padding (as fraction of raster's x pixel size) to add to left and right side of bounds.
- **pad_y** (*float*) – Amount of padding (as fraction of raster's y pixel size) to add to top and bottom of bounds.
- **north_up** (*optional*) – This parameter is ignored since version 1.2.1. A deprecation warning will be emitted in 1.3.0.

- **rotated** (*optional*) – This parameter is ignored since version 1.2.1. A deprecation warning will be emitted in 1.3.0.
- **pixel_precision** (*int or float, optional*) – Number of places of rounding precision or absolute precision for evaluating bounds of shapes.
- **boundless** (*bool, optional*) – Whether to allow a boundless window or not.

Returns

Return type *rasterio.windows.Window*

rasterio.features.**is_valid_geom**(*geom*)

Checks to see if geometry is a valid GeoJSON geometry type or GeometryCollection. Geometry must be GeoJSON or implement the geo interface.

Geometries must be non-empty, and have at least x, y coordinates.

Note: only the first coordinate is checked for validity.

Parameters **geom** (*an object that implements the geo interface or GeoJSON-like object*) –

Returns bool

Return type True if object is a valid GeoJSON geometry type

rasterio.features.**rasterize**(*shapes*, *out_shape=None*, *fill=0*, *out=None*, *transform=Affine(1.0, 0.0, 0.0, 0.0, 1.0, 0.0)*, *all_touched=False*, *merge_alg=<MergeAlg.replace: 'REPLACE'>*, *default_value=1*, *dtype=None*)

Return an image array with input geometries burned in.

Warnings will be raised for any invalid or empty geometries, and an exception will be raised if there are no valid shapes to rasterize.

Parameters

- **shapes** (iterable of (*geometry*, *value*) pairs or iterable over) – geometries. The *geometry* can either be an object that implements the geo interface or GeoJSON-like object. If no *value* is provided the *default_value* will be used. If *value* is *None* the *fill* value will be used.
- **out_shape** (*tuple or list with 2 integers*) – Shape of output numpy ndarray.
- **fill** (*int or float, optional*) – Used as fill value for all areas not covered by input geometries.
- **out** (*numpy ndarray, optional*) – Array of same shape and data type as *source* in which to store results.
- **transform** (*Affine transformation object, optional*) – Transformation from pixel coordinates of *source* to the coordinate system of the input *shapes*. See the *transform* property of dataset objects.
- **all_touched** (*boolean, optional*) – If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in.
- **merge_alg** (MergeAlg, *optional*) –

  Merge algorithm to use. One of:

  MergeAlg.replace (default): the new value will overwrite the existing value.

  MergeAlg.add: the new value will be added to the existing raster.

- **default_value** (*int or float, optional*) – Used as value for all geometries, if not provided in *shapes*.
- **dtype** (*rasterio or numpy data type, optional*) – Used as data type for results, if *out* is not provided.

Returns If *out* was not None then *out* is returned, it will have been modified in-place. If *out* was None, this will be a new array.

Return type numpy ndarray

Notes

Valid data types for *fill*, *default_value*, *out*, *dtype* and shape values are "int16", "int32", "uint8", "uint16", "uint32", "float32", and "float64".

This function requires significant memory resources. The shapes iterator will be materialized to a Python list and another C copy of that list will be made. The *out* array will be copied and additional temporary raster memory equal to 2x the smaller of *out* data or GDAL's max cache size (controlled by GDAL_CACHEMAX, default is 5% of the computer's physical memory) is required.

If GDAL max cache size is smaller than the output data, the array of shapes will be iterated multiple times. Performance is thus a linear function of buffer size. For maximum speed, ensure that GDAL_CACHEMAX is larger than the size of *out* or *out_shape*.

rasterio.features.**shapes**(*source*, *mask=None*, *connectivity=4*, *transform=Affine(1.0, 0.0, 0.0, 0.0, 1.0, 0.0)*)

Get shapes and values of connected regions in a dataset or array.

Parameters

- **source** (*array, dataset object, Band, or tuple(dataset, bidx)*) – Data type must be one of rasterio.int16, rasterio.int32, rasterio.uint8, rasterio.uint16, or rasterio.float32.
- **mask** (*numpy ndarray or rasterio Band object, optional*) – Must evaluate to bool (**rasterio.bool_** or rasterio.uint8). Values of False or 0 will be excluded from feature generation. Note well that this is the inverse sense from Numpy's, where a mask value of True indicates invalid data in an array. If *source* is a Numpy masked array and *mask* is None, the source's mask will be inverted and used in place of *mask*.
- **connectivity** (*int, optional*) – Use 4 or 8 pixel connectivity for grouping pixels into features
- **transform** (*Affine transformation, optional*) – If not provided, feature coordinates will be generated based on pixel coordinates

Yields *polygon, value* – A pair of (polygon, value) for each feature found in the image. Polygons are GeoJSON-like dicts and the values are the associated value from the image, in the data type of the image. Note: due to floating point precision issues, values returned from a floating point image may not exactly match the original values.

Notes

The amount of memory used by this algorithm is proportional to the number and complexity of polygons produced. This algorithm is most appropriate for simple thematic data. Data with high pixel-to-pixel variability, such as imagery, may produce one polygon per pixel and consume large amounts of memory.

Because the low-level implementation uses either an int32 or float32 buffer, uint32 and float64 data cannot be operated on without truncation issues.

rasterio.features.**sieve**(*source*, *size*, *out=None*, *mask=None*, *connectivity=4*)

Replace small polygons in *source* with value of their largest neighbor.

Polygons are found for each set of neighboring pixels of the same value.

Parameters

- **source** (*array or dataset object opened in 'r' mode or Band or tuple(dataset, bidx)*) – Must be of type rasterio.int16, rasterio.int32, rasterio.uint8, rasterio.uint16, or rasterio.float32
- **size** (*int*) – minimum polygon size (number of pixels) to retain.
- **out** (*numpy ndarray, optional*) – Array of same shape and data type as *source* in which to store results.
- **mask** (*numpy ndarray or rasterio Band object, optional*) – Values of False or 0 will be excluded from feature generation Must evaluate to bool (**rasterio.bool_** or rasterio.uint8)
- **connectivity** (*int, optional*) – Use 4 or 8 pixel connectivity for grouping pixels into features

Returns **out** – Result

Return type numpy ndarray

Notes

GDAL only supports values that can be cast to 32-bit integers for this operation.

The amount of memory used by this algorithm is proportional to the number and complexity of polygons found in the image. This algorithm is most appropriate for simple thematic data. Data with high pixel-to-pixel variability, such as imagery, may produce one polygon per pixel and consume large amounts of memory.

rasterio.fill module

Fill holes in raster dataset by interpolation from the edges.

rasterio.fill.**fillnodata**(*image*, *mask=None*, *max_search_distance=100.0*, *smoothing_iterations=0*)

Fill holes in raster data by interpolation

This algorithm will interpolate values for all designated nodata pixels (marked by zeros in *mask*). For each pixel a four direction conic search is done to find values to interpolate from (using inverse distance weighting). Once all values are interpolated, zero or more smoothing iterations (3x3 average filters on interpolated pixels) are applied to smooth out artifacts.

This algorithm is generally suitable for interpolating missing regions of fairly continuously varying rasters (such as elevation models for instance). It is also suitable for filling small holes and cracks in more irregularly varying images (like aerial photos). It is generally not so great for interpolating a raster from sparse point data.

Parameters

- **image** (*numpy ndarray*) – The source image with holes to be filled. If a MaskedArray, the inverse of its mask will define the pixels to be filled – unless the `mask` argument is not None (see below).`
- **mask** (*numpy ndarray or None*) – A mask band indicating which pixels to interpolate. Pixels to interpolate into are indicated by the value 0. Values > 0 indicate areas to use during interpolation. Must be same shape as image. This array always takes precedence over the image's mask (see above). If None, the inverse of the image's mask will be used if available.
- **max_search_distance** (*float, optional*) – The maximum number of pixels to search in all directions to find values to interpolate from. The default is 100.
- **smoothing_iterations** (*integer, optional*) – The number of 3x3 smoothing filter passes to run. The default is 0.

Returns **out** – The filled raster array.

Return type numpy ndarray

rasterio.io module

Classes capable of reading and writing datasets

Instances of these classes are called dataset objects.

class rasterio.io.**BufferedDatasetWriter**

Bases: *rasterio._io.BufferedDatasetWriterBase*, *rasterio.windows.WindowMethodsMixin*, *rasterio.transform.TransformMethodsMixin*

Maintains data and metadata in a buffer, writing to disk or network only when *close()* is called.

This allows incremental updates to datasets using formats that don't otherwise support updates, such as JPEG.

block_shapes

An ordered list of block shapes for each bands

Shapes are tuples and have the same ordering as the dataset's shape: (count of image rows, count of image columns).

Returns

Return type list

block_size()

Returns the size in bytes of a particular block

Only useful for TIFF formatted datasets.

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type int

block_window()

Returns the window for a particular block

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type *Window*

block_windows()

Iterator over a band's blocks and their windows

The primary use of this method is to obtain windows to pass to *read()* for highly efficient access to raster block data.

The positional parameter *bidx* takes the index (starting at 1) of the desired band. This iterator yields blocks "left to right" and "top to bottom" and is similar to Python's `enumerate()` in that the first element is the block index and the second is the dataset window.

Blocks are built-in to a dataset and describe how pixels are grouped within each band and provide a mechanism for efficient I/O. A window is a range of pixels within a single band defined by row start, row stop, column start, and column stop. For example, `((0, 2), (0, 2))` defines a `2 x 2` window at the upper left corner of a raster band. Blocks are referenced by an `(i, j)` tuple where `(0, 0)` would be a band's upper left block.

Raster I/O is performed at the block level, so accessing a window spanning multiple rows in a striped raster requires reading each row. Accessing a `2 x 2` window at the center of a `1800 x 3600` image requires reading 2 rows, or 7200 pixels just to get the target 4. The same image with internal `256 x 256` blocks would require reading at least 1 block (if the window entire window falls within a single block) and at most 4 blocks, or at least 512 pixels and at most 2048.

Given an image that is `512 x 512` with blocks that are `256 x 256`, its blocks and windows would look like:

```
Blocks:

        0       256     512
      0 +--------+--------+
        |        |        |
        | (0, 0) | (0, 1) |
        |        |        |
    256 +--------+--------+
        |        |        |
        | (1, 0) | (1, 1) |
        |        |        |
    512 +--------+--------+


Windows:

    UL: ((0, 256), (0, 256))
    UR: ((0, 256), (256, 512))
    LL: ((256, 512), (0, 256))
    LR: ((256, 512), (256, 512))
```

Parameters **bidx** (*int, optional*) – The band index (using 1-based indexing) from which to extract windows. A value less than 1 uses the first band if all bands have homogeneous windows and raises an exception otherwise.

Yields *block, window*

bounds
Returns the lower left and upper right bounds of the dataset in the units of its coordinate reference system.

The returned value is a tuple: (lower left x, lower left y, upper right x, upper right y)

build_overviews()
Build overviews at one or more decimation factors for all bands of the dataset.

checksum()
Compute an integer checksum for the stored band

Parameters

- **bidx** (*int*) – The band's index (1-indexed).
- **window** (*tuple, optional*) – A window of the band. Default is the entire extent of the band.

Returns

Return type An int.

close()
Close the dataset

closed
Test if the dataset is closed

Returns

Return type bool

colorinterp
Returns a sequence of `ColorInterp.<enum>` representing color interpretation in band order.

To set color interpretation, provide a sequence of `ColorInterp.<enum>`:

import rasterio from rasterio.enums import ColorInterp

with rasterio.open('rgba.tif', 'r+') as src:

src.colorinterp = (ColorInterp.red, ColorInterp.green, ColorInterp.blue, ColorInterp.alpha)

Returns

Return type tuple

colormap()
Returns a dict containing the colormap for a band or None.

compression

count
The number of raster bands in the dataset

Returns

Return type int

crs
The dataset's coordinate reference system

In setting this property, the value may be a CRS object or an EPSG:nnnn or WKT string.

Returns

Return type *CRS*

dataset_mask()
Get the dataset's 2D valid data mask.

Parameters

- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot be combined with *out_shape*.

- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot be combined with *out*.

- **window**(*a pair (tuple) of pairs of ints or Window, optional*)– The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns The dtype of this array is uint8. 0 = nodata, 255 = valid data.

Return type Numpy ndarray or a view on a Numpy ndarray

Notes

Note: as with Numpy ufuncs, an object is returned even if you use the optional *out* argument and the return value shall be preferentially used by callers.

The dataset mask is calculated based on the individual band masks according to the following logic, in order of precedence:

1. If a .msk file, dataset-wide alpha, or internal mask exists it will be used for the dataset mask.
2. Else if the dataset is a 4-band 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 255.

Note that this differs from read_masks and GDAL RFC15 in that it applies per-dataset, not per-band (see https://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask)

descriptions
Descriptions for each dataset band

To set descriptions, one for each band is required.

Returns

Return type list of str

driver

dtypes
The data types of each band in index order

Returns

Return type list of str

files
Returns a sequence of files associated with the dataset.

Returns

Return type tuple

gcps
ground control points and their coordinate reference system.

This property is a 2-tuple, or pair: (gcps, crs).

gcps [list of GroundControlPoint] Zero or more ground control points.

crs: CRS The coordinate reference system of the ground control points.

get_gcps()
Get GCPs and their associated CRS.

get_nodatavals()

get_tag_item()
Returns tag item value

Parameters

- **ns** (*str*) – The key for the metadata item to fetch.
- **dm** (*str*) – The domain to fetch for.
- **bidx** (*int*) – Band index, starting with 1.
- **ovr** (*int*) – Overview level

Returns

Return type str

get_transform()
Returns a GDAL geotransform in its native form.

height

index(*x*, *y*, *op=<built-in function floor>*, *precision=None*)
Returns the (row, col) index of the pixel containing (x, y) given a coordinate reference system.

Use an epsilon, magnitude determined by the precision parameter and sign determined by the op function:

positive for floor, negative for ceil.

Parameters

- **x** (*float*) – x value in coordinate reference system
- **y** (*float*) – y value in coordinate reference system

- **op** (*function, optional (default: math.floor)*) – Function to convert fractional pixels to whole numbers (floor, ceiling, round)
- **precision** (*int, optional (default: None)*) – Decimal places of precision in indexing, as in *round()*.

Returns (row index, col index)

Return type tuple

indexes

The 1-based indexes of each band in the dataset

For a 3-band dataset, this property will be `[1, 2, 3]`.

Returns

Return type list of int

interleaving

is_tiled

lnglat()

mask_flag_enums

Sets of flags describing the sources of band masks.

all_valid: There are no invalid pixels, all mask values will be

255. When used this will normally be the only flag set.

per_dataset: The mask band is shared between all bands on the dataset.

alpha: The mask band is actually an alpha band and may have values other than 0 and 255.

nodata: Indicates the mask is actually being generated from nodata values (mutually exclusive of "alpha").

Returns One list of rasterio.enums.MaskFlags members per band.

Return type list [, list*]

Examples

For a 3 band dataset that has masks derived from nodata values:

```
>>> dataset.mask_flag_enums
([<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>])
>>> band1_flags = dataset.mask_flag_enums[0]
>>> rasterio.enums.MaskFlags.nodata in band1_flags
True
>>> rasterio.enums.MaskFlags.alpha in band1_flags
False
```

meta

The basic metadata of this dataset.

mode

name

nodata

The dataset's single nodata value

Notes

May be set.

Returns

Return type float

nodatavals
Nodata values for each band

Notes

This may not be set.

Returns

Return type list of float

offsets
Raster offset for each dataset band

To set offsets, one for each band is required.

Returns

Return type list of float

options

overviews()

photometric

profile
Basic metadata and creation options of this dataset.

May be passed as keyword arguments to *rasterio.open()* to create a clone of this dataset.

read()
Read a dataset's raw pixels as an N-d array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array into which data will be placed. If the height and width of *out* differ from that of the specified window (see below), the raster image will be decimated or replicated using the specified resampling method (also see below).

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  This parameter cannot be combined with *out_shape*.
- **out_dtype** (*str or numpy dtype*) – The desired output data type. For example: 'uint8' or rasterio.uint16.

- **out_shape** (*tuple, optional*) – A tuple describing the shape of a new output array. See *out* (above) for notes on image decimation and replication.

  Cannot combined with *out*.

- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **masked** (*bool, optional*) – If *masked* is *True* the return value will be a masked array. Otherwise (the default) the return value will be a regular array. Masks will be exactly the inverse of the GDAL RFC 15 conforming arrays returned by read_masks().

- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

- **fill_value** (*scalar*) – Fill value applied in the *boundless=True* case only. Like the fill_value of numpy.ma.MaskedArray, should be value valid for the dataset's data type.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_crs()
Return the GDAL dataset's stored CRS

read_masks()
Read raster band masks as a multidimensional array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.

- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot combine with *out_shape*.

- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot combined with *out*.

- **window** (*a pair (tuple) of pairs of ints or* Window*, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_transform()
Return the stored GDAL GeoTransform

res
Returns the (width, height) of pixels in the units of its coordinate reference system.

rpcs
Rational polynomial coefficients mapping between pixel and geodetic coordinates.

This property is a dict-like object.

rpcs : RPC instance containing coefficients. Empty if dataset does not have any metadata in the "RPC" domain.

sample()
Get the values of a dataset at certain positions

Values are from the nearest pixel. They are not interpolated.

Parameters

- **xy** (*iterable*) – Pairs of x, y coordinates (floats) in the dataset's reference system.
- **indexes** (*int or list of int*) – Indexes of dataset bands to sample.
- **masked** (*bool, default: False*) – Whether to mask samples that fall outside the extent of the dataset.

Returns Arrays of length equal to the number of specified indexes containing the dataset values for the bands corresponding to those indexes.

Return type iterable

scales
Raster scale for each dataset band

To set scales, one for each band is required.

Returns

Return type list of float

set_band_description()
Sets the description of a dataset band.

Parameters

- **bidx** (*int*) – Index of the band (starting with 1).
- **value** (*string*) – A description of the band.

Returns

Return type None

set_band_unit()
Sets the unit of measure of a dataset band.

Parameters

- **bidx** (*int*) – Index of the band (starting with 1).
- **value** (*string*) – A label for the band's unit of measure such as 'meters' or 'degC'. See the Pint project for a suggested list of units.

Returns

Return type None

shape

start()
Start the dataset's life cycle

stop()
Close the GDAL dataset handle

subdatasets
Sequence of subdatasets

tag_namespaces()
Get a list of the dataset's metadata domains.

Returned items may be passed as *ns* to the tags method.

Parameters

- **int** (*bidx*) – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.
- **optional** – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.

Returns

Return type list of str

tags()
Returns a dict containing copies of the dataset or band's tags.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the tags of a specific band. The optional ns argument can be used to select a namespace other than the default.

transform
The dataset's georeferencing transformation matrix

This transform maps pixel row/column coordinates to coordinates in the dataset's coordinate reference system.

Returns

Return type Affine

units
one units string for each dataset band

Possible values include 'meters' or 'degC'. See the Pint project for a suggested list of units.

To set units, one for each band is required.

Returns

Return type list of str

Type A list of str

update_tags()
Updates the tags of a dataset or one of its bands.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the dataset band. The optional ns argument can be used to select a namespace other than the default.

width

window(*left*, *bottom*, *right*, *top*, *precision=None*)
Get the window corresponding to the bounding coordinates.

The resulting window is not cropped to the row and column limits of the dataset.

Parameters

- **left** (*float*) – Left (west) bounding coordinate
- **bottom** (*float*) – Bottom (south) bounding coordinate
- **right** (*float*) – Right (east) bounding coordinate
- **top** (*float*) – Top (north) bounding coordinate
- **precision** (*int, optional*) – Number of decimal points of precision when computing inverse transform.

Returns **window**

Return type *Window*

window_bounds(*window*)
Get the bounds of a window

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **bounds** – x_min, y_min, x_max, y_max for the given window

Return type tuple

window_transform(*window*)
Get the affine transform for a dataset window.

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **transform** – The affine transform matrix for the given window

Return type Affine

write()
Write the arr array into indexed bands of the dataset.

If *indexes* is a list, the src must be a 3D array of matching shape. If an int, the src must be a 2D array.

See *read()* for usage of the optional *window* argument.

write_band()
Write the src array into the *bidx* band.

Band indexes begin with 1: read_band(1) returns the first band.

The optional *window* argument takes a tuple like:

((row_start, row_stop), (col_start, col_stop))

specifying a raster subset to write into.

write_colormap()
Write a colormap for a band to the dataset.

write_mask()
Write the valid data mask src array into the dataset's band mask.

The optional *window* argument takes a tuple like:

((row_start, row_stop), (col_start, col_stop))

specifying a raster subset to write into.

write_transform()

xy(*row*, *col*, *offset='center'*)
Returns the coordinates `(x, y)` of a pixel at *row* and *col*. The pixel's center is returned by default, but a corner can be returned by setting *offset* to one of *ul, ur, ll, lr*.

Parameters

- **row** (*int*) – Pixel row.
- **col** (*int*) – Pixel column.
- **offset** (*str, optional*) – Determines if the returned coordinates are for the center of the pixel or for a corner.

Returns `(x, y)`

Return type tuple

class rasterio.io.**DatasetReader**
Bases: *rasterio._io.DatasetReaderBase*, *rasterio.windows.WindowMethodsMixin*, *rasterio.transform.TransformMethodsMixin*

An unbuffered data and metadata reader

block_shapes
An ordered list of block shapes for each bands

Shapes are tuples and have the same ordering as the dataset's shape: (count of image rows, count of image columns).

Returns

Return type list

block_size()
Returns the size in bytes of a particular block

Only useful for TIFF formatted datasets.

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type int

block_window()
Returns the window for a particular block

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type *Window*

block_windows()
Iterator over a band's blocks and their windows

The primary use of this method is to obtain windows to pass to *read()* for highly efficient access to raster block data.

The positional parameter *bidx* takes the index (starting at 1) of the desired band. This iterator yields blocks "left to right" and "top to bottom" and is similar to Python's `enumerate()` in that the first element is the block index and the second is the dataset window.

Blocks are built-in to a dataset and describe how pixels are grouped within each band and provide a mechanism for efficient I/O. A window is a range of pixels within a single band defined by row start, row stop, column start, and column stop. For example, `((0, 2), (0, 2))` defines a `2 x 2` window at the upper left corner of a raster band. Blocks are referenced by an `(i, j)` tuple where `(0, 0)` would be a band's upper left block.

Raster I/O is performed at the block level, so accessing a window spanning multiple rows in a striped raster requires reading each row. Accessing a `2 x 2` window at the center of a `1800 x 3600` image requires reading 2 rows, or 7200 pixels just to get the target 4. The same image with internal `256 x 256` blocks would require reading at least 1 block (if the window entire window falls within a single block) and at most 4 blocks, or at least 512 pixels and at most 2048.

Given an image that is `512 x 512` with blocks that are `256 x 256`, its blocks and windows would look like:

```
Blocks:

        0       256     512
      0 +--------+--------+
        |        |        |
        | (0, 0) | (0, 1) |
        |        |        |
```

(continues on next page)

(continued from previous page)

```
    256 +--------+--------+
        |        |        |
        | (1, 0) | (1, 1) |
        |        |        |
    512 +--------+--------+


Windows:

    UL: ((0, 256), (0, 256))
    UR: ((0, 256), (256, 512))
    LL: ((256, 512), (0, 256))
    LR: ((256, 512), (256, 512))
```

Parameters **bidx** (*int, optional*) – The band index (using 1-based indexing) from which to extract windows. A value less than 1 uses the first band if all bands have homogeneous windows and raises an exception otherwise.

Yields *block, window*

bounds

Returns the lower left and upper right bounds of the dataset in the units of its coordinate reference system.

The returned value is a tuple: (lower left x, lower left y, upper right x, upper right y)

checksum()

Compute an integer checksum for the stored band

Parameters

- **bidx** (*int*) – The band's index (1-indexed).
- **window** (*tuple, optional*) – A window of the band. Default is the entire extent of the band.

Returns

Return type An int.

close()

Close the dataset

closed

Test if the dataset is closed

Returns

Return type bool

colorinterp

Returns a sequence of `ColorInterp.<enum>` representing color interpretation in band order.

To set color interpretation, provide a sequence of `ColorInterp.<enum>`:

import rasterio from rasterio.enums import ColorInterp

with rasterio.open('rgba.tif', 'r+') as src:

src.colorinterp = (ColorInterp.red, ColorInterp.green, ColorInterp.blue, ColorInterp.alpha)

Returns

Return type tuple

colormap()
Returns a dict containing the colormap for a band or None.

compression

count
The number of raster bands in the dataset

Returns

Return type int

crs
The dataset's coordinate reference system

In setting this property, the value may be a CRS object or an EPSG:nnnn or WKT string.

Returns

Return type *CRS*

dataset_mask()
Get the dataset's 2D valid data mask.

Parameters

- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot be combined with *out_shape*.

- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot be combined with *out*.

- **window**(*a pair (tuple) of pairs of ints or Window, optional*)– The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns The dtype of this array is uint8. 0 = nodata, 255 = valid data.

Return type Numpy ndarray or a view on a Numpy ndarray

Notes

Note: as with Numpy ufuncs, an object is returned even if you use the optional *out* argument and the return value shall be preferentially used by callers.

The dataset mask is calculated based on the individual band masks according to the following logic, in order of precedence:

1. If a .msk file, dataset-wide alpha, or internal mask exists it will be used for the dataset mask.
2. Else if the dataset is a 4-band 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 255.

Note that this differs from read_masks and GDAL RFC15 in that it applies per-dataset, not per-band (see https://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask)

descriptions
Descriptions for each dataset band

To set descriptions, one for each band is required.

Returns

Return type list of str

driver

dtypes
The data types of each band in index order

Returns

Return type list of str

files
Returns a sequence of files associated with the dataset.

Returns

Return type tuple

gcps
ground control points and their coordinate reference system.

This property is a 2-tuple, or pair: (gcps, crs).

gcps [list of GroundControlPoint] Zero or more ground control points.

crs: CRS The coordinate reference system of the ground control points.

get_gcps()
Get GCPs and their associated CRS.

get_nodatavals()

get_tag_item()
Returns tag item value

Parameters

- **ns** (*str*) – The key for the metadata item to fetch.
- **dm** (*str*) – The domain to fetch for.
- **bidx** (*int*) – Band index, starting with 1.

- **ovr** (*int*) – Overview level

Returns

Return type str

get_transform()
Returns a GDAL geotransform in its native form.

height

index(*x*, *y*, *op=<built-in function floor>*, *precision=None*)
Returns the (row, col) index of the pixel containing (x, y) given a coordinate reference system.

Use an epsilon, magnitude determined by the precision parameter and sign determined by the op function:

positive for floor, negative for ceil.

Parameters

- **x** (*float*) – x value in coordinate reference system
- **y** (*float*) – y value in coordinate reference system
- **op** (*function, optional (default: math.floor)*) – Function to convert fractional pixels to whole numbers (floor, ceiling, round)
- **precision** (*int, optional (default: None)*) – Decimal places of precision in indexing, as in *round()*.

Returns (row index, col index)

Return type tuple

indexes
The 1-based indexes of each band in the dataset

For a 3-band dataset, this property will be `[1, 2, 3]`.

Returns

Return type list of int

interleaving

is_tiled

lnglat()

mask_flag_enums
Sets of flags describing the sources of band masks.

all_valid: There are no invalid pixels, all mask values will be

255. When used this will normally be the only flag set.

per_dataset: The mask band is shared between all bands on the dataset.

alpha: The mask band is actually an alpha band and may have values other than 0 and 255.

nodata: Indicates the mask is actually being generated from nodata values (mutually exclusive of "alpha").

Returns One list of rasterio.enums.MaskFlags members per band.

Return type list [, list*]

Examples

For a 3 band dataset that has masks derived from nodata values:

```
>>> dataset.mask_flag_enums
([<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>])
>>> band1_flags = dataset.mask_flag_enums[0]
>>> rasterio.enums.MaskFlags.nodata in band1_flags
True
>>> rasterio.enums.MaskFlags.alpha in band1_flags
False
```

meta
The basic metadata of this dataset.

mode

name

nodata
The dataset's single nodata value

Notes

May be set.

Returns

Return type float

nodatavals
Nodata values for each band

Notes

This may not be set.

Returns

Return type list of float

offsets
Raster offset for each dataset band

To set offsets, one for each band is required.

Returns

Return type list of float

options

overviews()

photometric

profile
Basic metadata and creation options of this dataset.

May be passed as keyword arguments to *rasterio.open()* to create a clone of this dataset.

read()
Read a dataset's raw pixels as an N-d array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array into which data will be placed. If the height and width of *out* differ from that of the specified window (see below), the raster image will be decimated or replicated using the specified resampling method (also see below).

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  This parameter cannot be combined with *out_shape*.
- **out_dtype** (*str or numpy dtype*) – The desired output data type. For example: 'uint8' or rasterio.uint16.
- **out_shape** (*tuple, optional*) – A tuple describing the shape of a new output array. See *out* (above) for notes on image decimation and replication.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **masked** (*bool, optional*) – If *masked* is *True* the return value will be a masked array. Otherwise (the default) the return value will be a regular array. Masks will be exactly the inverse of the GDAL RFC 15 conforming arrays returned by read_masks().
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.
- **fill_value** (*scalar*) – Fill value applied in the *boundless=True* case only. Like the fill_value of numpy.ma.MaskedArray, should be value valid for the dataset's data type.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_crs()
Return the GDAL dataset's stored CRS

read_masks()
Read raster band masks as a multidimensional array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot combine with *out_shape*.
- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_transform()
Return the stored GDAL GeoTransform

res
Returns the (width, height) of pixels in the units of its coordinate reference system.

rpcs
Rational polynomial coefficients mapping between pixel and geodetic coordinates.

This property is a dict-like object.

rpcs : RPC instance containing coefficients. Empty if dataset does not have any metadata in the "RPC" domain.

sample()
Get the values of a dataset at certain positions

Values are from the nearest pixel. They are not interpolated.

Parameters

- **xy** (*iterable*) – Pairs of x, y coordinates (floats) in the dataset's reference system.
- **indexes** (*int or list of int*) – Indexes of dataset bands to sample.
- **masked** (*bool, default: False*) – Whether to mask samples that fall outside the extent of the dataset.

Returns Arrays of length equal to the number of specified indexes containing the dataset values for the bands corresponding to those indexes.

Return type iterable

scales
Raster scale for each dataset band

To set scales, one for each band is required.

Returns

Return type list of float

shape

start()
Start the dataset's life cycle

stop()
Close the GDAL dataset handle

subdatasets
Sequence of subdatasets

tag_namespaces()
Get a list of the dataset's metadata domains.

Returned items may be passed as *ns* to the tags method.

Parameters

- **int** (*bidx*) – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.
- **optional** – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.

Returns

Return type list of str

tags()
Returns a dict containing copies of the dataset or band's tags.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the tags of a specific band. The optional ns argument can be used to select a namespace other than the default.

transform
The dataset's georeferencing transformation matrix

This transform maps pixel row/column coordinates to coordinates in the dataset's coordinate reference system.

Returns

Return type Affine

units
one units string for each dataset band

Possible values include 'meters' or 'degC'. See the Pint project for a suggested list of units.

To set units, one for each band is required.

Returns

Return type list of str

Type A list of str

width

window(*left*, *bottom*, *right*, *top*, *precision=None*)
Get the window corresponding to the bounding coordinates.

The resulting window is not cropped to the row and column limits of the dataset.

Parameters

- **left** (*float*) – Left (west) bounding coordinate
- **bottom** (*float*) – Bottom (south) bounding coordinate
- **right** (*float*) – Right (east) bounding coordinate
- **top** (*float*) – Top (north) bounding coordinate
- **precision** (*int, optional*) – Number of decimal points of precision when computing inverse transform.

Returns **window**

Return type *Window*

window_bounds(*window*)
Get the bounds of a window

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **bounds** – x_min, y_min, x_max, y_max for the given window

Return type tuple

window_transform(*window*)
Get the affine transform for a dataset window.

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **transform** – The affine transform matrix for the given window

Return type Affine

write_transform()

xy(*row*, *col*, *offset='center'*)
Returns the coordinates (x, y) of a pixel at *row* and *col*. The pixel's center is returned by default, but a corner can be returned by setting *offset* to one of *ul, ur, ll, lr*.

Parameters

- **row** (*int*) – Pixel row.

- **col** (*int*) – Pixel column.
- **offset** (*str, optional*) – Determines if the returned coordinates are for the center of the pixel or for a corner.

Returns (x, y)

Return type tuple

class rasterio.io.**DatasetWriter**

Bases: *rasterio._io.DatasetWriterBase*, *rasterio.windows.WindowMethodsMixin*, *rasterio.transform.TransformMethodsMixin*

An unbuffered data and metadata writer. Its methods write data directly to disk.

block_shapes

An ordered list of block shapes for each bands

Shapes are tuples and have the same ordering as the dataset's shape: (count of image rows, count of image columns).

Returns

Return type list

block_size()

Returns the size in bytes of a particular block

Only useful for TIFF formatted datasets.

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type int

block_window()

Returns the window for a particular block

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type *Window*

block_windows()

Iterator over a band's blocks and their windows

The primary use of this method is to obtain windows to pass to *read()* for highly efficient access to raster block data.

The positional parameter *bidx* takes the index (starting at 1) of the desired band. This iterator yields blocks "left to right" and "top to bottom" and is similar to Python's enumerate() in that the first element is the block index and the second is the dataset window.

Blocks are built-in to a dataset and describe how pixels are grouped within each band and provide a mechanism for efficient I/O. A window is a range of pixels within a single band defined by row start, row stop, column start, and column stop. For example, `((0, 2), (0, 2))` defines a `2 x 2` window at the upper left corner of a raster band. Blocks are referenced by an `(i, j)` tuple where `(0, 0)` would be a band's upper left block.

Raster I/O is performed at the block level, so accessing a window spanning multiple rows in a striped raster requires reading each row. Accessing a `2 x 2` window at the center of a `1800 x 3600` image requires reading 2 rows, or 7200 pixels just to get the target 4. The same image with internal `256 x 256` blocks would require reading at least 1 block (if the window entire window falls within a single block) and at most 4 blocks, or at least 512 pixels and at most 2048.

Given an image that is `512 x 512` with blocks that are `256 x 256`, its blocks and windows would look like:

```
Blocks:

        0       256     512
      0 +--------+--------+
        |        |        |
        | (0, 0) | (0, 1) |
        |        |        |
    256 +--------+--------+
        |        |        |
        | (1, 0) | (1, 1) |
        |        |        |
    512 +--------+--------+


Windows:

    UL: ((0, 256), (0, 256))
    UR: ((0, 256), (256, 512))
    LL: ((256, 512), (0, 256))
    LR: ((256, 512), (256, 512))
```

Parameters **bidx** (*int, optional*) – The band index (using 1-based indexing) from which to extract windows. A value less than 1 uses the first band if all bands have homogeneous windows and raises an exception otherwise.

Yields *block, window*

bounds
Returns the lower left and upper right bounds of the dataset in the units of its coordinate reference system.

The returned value is a tuple: (lower left x, lower left y, upper right x, upper right y)

build_overviews()
Build overviews at one or more decimation factors for all bands of the dataset.

checksum()
Compute an integer checksum for the stored band

Parameters

- **bidx** (*int*) – The band's index (1-indexed).
- **window** (*tuple, optional*) – A window of the band. Default is the entire extent of the band.

Returns

Return type An int.

close()
Close the dataset

closed
Test if the dataset is closed

Returns

Return type bool

colorinterp
Returns a sequence of `ColorInterp.<enum>` representing color interpretation in band order.

To set color interpretation, provide a sequence of `ColorInterp.<enum>`:

import rasterio from rasterio.enums import ColorInterp

with rasterio.open('rgba.tif', 'r+') as src:

src.colorinterp = (ColorInterp.red, ColorInterp.green, ColorInterp.blue, ColorInterp.alpha)

Returns

Return type tuple

colormap()
Returns a dict containing the colormap for a band or None.

compression

count
The number of raster bands in the dataset

Returns

Return type int

crs
The dataset's coordinate reference system

In setting this property, the value may be a CRS object or an EPSG:nnnn or WKT string.

Returns

Return type *CRS*

dataset_mask()
Get the dataset's 2D valid data mask.

Parameters

- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot be combined with *out_shape*.

- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot be combined with *out*.

- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns The dtype of this array is uint8. 0 = nodata, 255 = valid data.

Return type Numpy ndarray or a view on a Numpy ndarray

Notes

Note: as with Numpy ufuncs, an object is returned even if you use the optional *out* argument and the return value shall be preferentially used by callers.

The dataset mask is calculated based on the individual band masks according to the following logic, in order of precedence:

1. If a .msk file, dataset-wide alpha, or internal mask exists it will be used for the dataset mask.
2. Else if the dataset is a 4-band 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 255.

Note that this differs from read_masks and GDAL RFC15 in that it applies per-dataset, not per-band (see https://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask)

descriptions
Descriptions for each dataset band

To set descriptions, one for each band is required.

Returns

Return type list of str

driver

dtypes
The data types of each band in index order

Returns

Return type list of str

files
Returns a sequence of files associated with the dataset.

Returns

Return type tuple

gcps
ground control points and their coordinate reference system.

This property is a 2-tuple, or pair: (gcps, crs).

gcps [list of GroundControlPoint] Zero or more ground control points.

crs: CRS The coordinate reference system of the ground control points.

get_gcps()
Get GCPs and their associated CRS.

get_nodatavals()

get_tag_item()
Returns tag item value

Parameters

- **ns** (*str*) – The key for the metadata item to fetch.
- **dm** (*str*) – The domain to fetch for.
- **bidx** (*int*) – Band index, starting with 1.
- **ovr** (*int*) – Overview level

Returns

Return type str

get_transform()
Returns a GDAL geotransform in its native form.

height

index(*x*, *y*, *op=<built-in function floor>*, *precision=None*)
Returns the (row, col) index of the pixel containing (x, y) given a coordinate reference system.

Use an epsilon, magnitude determined by the precision parameter and sign determined by the op function:

positive for floor, negative for ceil.

Parameters

- **x** (*float*) – x value in coordinate reference system
- **y** (*float*) – y value in coordinate reference system
- **op** (*function, optional (default: math.floor)*) – Function to convert fractional pixels to whole numbers (floor, ceiling, round)
- **precision** (*int, optional (default: None)*) – Decimal places of precision in indexing, as in *round()*.

Returns (row index, col index)

Return type tuple

indexes
The 1-based indexes of each band in the dataset

For a 3-band dataset, this property will be `[1, 2, 3]`.

Returns

Return type list of int

interleaving

is_tiled

lnglat()

mask_flag_enums

Sets of flags describing the sources of band masks.

all_valid: There are no invalid pixels, all mask values will be

255. When used this will normally be the only flag set.

per_dataset: The mask band is shared between all bands on the dataset.

alpha: The mask band is actually an alpha band and may have values other than 0 and 255.

nodata: Indicates the mask is actually being generated from nodata values (mutually exclusive of "alpha").

Returns One list of rasterio.enums.MaskFlags members per band.

Return type list [, list*]

Examples

For a 3 band dataset that has masks derived from nodata values:

```
>>> dataset.mask_flag_enums
([<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>])
>>> band1_flags = dataset.mask_flag_enums[0]
>>> rasterio.enums.MaskFlags.nodata in band1_flags
True
>>> rasterio.enums.MaskFlags.alpha in band1_flags
False
```

meta

The basic metadata of this dataset.

mode

name

nodata

The dataset's single nodata value

Notes

May be set.

Returns

Return type float

nodatavals

Nodata values for each band

Notes

This may not be set.

Returns

Return type list of float

offsets
Raster offset for each dataset band

To set offsets, one for each band is required.

Returns

Return type list of float

options

overviews()

photometric

profile
Basic metadata and creation options of this dataset.

May be passed as keyword arguments to *rasterio.open()* to create a clone of this dataset.

read()
Read a dataset's raw pixels as an N-d array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array into which data will be placed. If the height and width of *out* differ from that of the specified window (see below), the raster image will be decimated or replicated using the specified resampling method (also see below).

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  This parameter cannot be combined with *out_shape*.
- **out_dtype** (*str or numpy dtype*) – The desired output data type. For example: 'uint8' or rasterio.uint16.
- **out_shape** (*tuple, optional*) – A tuple describing the shape of a new output array. See *out* (above) for notes on image decimation and replication.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **masked** (*bool, optional*) – If *masked* is *True* the return value will be a masked array. Otherwise (the default) the return value will be a regular array. Masks will be exactly the inverse of the GDAL RFC 15 conforming arrays returned by read_masks().
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.
- **fill_value** (*scalar*) – Fill value applied in the *boundless=True* case only. Like the fill_value of numpy.ma.MaskedArray, should be value valid for the dataset's data type.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_crs()
Return the GDAL dataset's stored CRS

read_masks()
Read raster band masks as a multidimensional array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  Cannot combine with *out_shape*.
- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot combined with *out*.
- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_transform()
Return the stored GDAL GeoTransform

res
Returns the (width, height) of pixels in the units of its coordinate reference system.

rpcs
Rational polynomial coefficients mapping between pixel and geodetic coordinates.

This property is a dict-like object.

rpcs : RPC instance containing coefficients. Empty if dataset does not have any metadata in the "RPC" domain.

sample()
Get the values of a dataset at certain positions

Values are from the nearest pixel. They are not interpolated.

Parameters

- **xy** (*iterable*) – Pairs of x, y coordinates (floats) in the dataset's reference system.
- **indexes** (*int or list of int*) – Indexes of dataset bands to sample.
- **masked** (*bool, default: False*) – Whether to mask samples that fall outside the extent of the dataset.

Returns Arrays of length equal to the number of specified indexes containing the dataset values for the bands corresponding to those indexes.

Return type iterable

scales
Raster scale for each dataset band

To set scales, one for each band is required.

Returns

Return type list of float

set_band_description()
Sets the description of a dataset band.

Parameters

- **bidx** (*int*) – Index of the band (starting with 1).
- **value** (*string*) – A description of the band.

Returns

Return type None

set_band_unit()
Sets the unit of measure of a dataset band.

Parameters

- **bidx** (*int*) – Index of the band (starting with 1).
- **value** (*string*) – A label for the band's unit of measure such as 'meters' or 'degC'. See the Pint project for a suggested list of units.

Returns

Return type None

shape

start()
Start the dataset's life cycle

stop()
Close the GDAL dataset handle

subdatasets
Sequence of subdatasets

tag_namespaces()
Get a list of the dataset's metadata domains.

Returned items may be passed as *ns* to the tags method.

Parameters

- **int** (*bidx*) – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.
- **optional** – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.

Returns

Return type list of str

tags()
Returns a dict containing copies of the dataset or band's tags.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the tags of a specific band. The optional ns argument can be used to select a namespace other than the default.

transform
The dataset's georeferencing transformation matrix

This transform maps pixel row/column coordinates to coordinates in the dataset's coordinate reference system.

Returns

Return type Affine

units
one units string for each dataset band

Possible values include 'meters' or 'degC'. See the Pint project for a suggested list of units.

To set units, one for each band is required.

Returns

Return type list of str

Type A list of str

update_tags()
Updates the tags of a dataset or one of its bands.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the dataset band. The optional ns argument can be used to select a namespace other than the default.

width

window(*left*, *bottom*, *right*, *top*, *precision=None*)
Get the window corresponding to the bounding coordinates.

The resulting window is not cropped to the row and column limits of the dataset.

Parameters

- **left** (*float*) – Left (west) bounding coordinate
- **bottom** (*float*) – Bottom (south) bounding coordinate
- **right** (*float*) – Right (east) bounding coordinate
- **top** (*float*) – Top (north) bounding coordinate
- **precision** (*int, optional*) – Number of decimal points of precision when computing inverse transform.

Returns **window**

Return type *Window*

window_bounds(*window*)
Get the bounds of a window

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **bounds** – x_min, y_min, x_max, y_max for the given window

Return type tuple

window_transform(*window*)
Get the affine transform for a dataset window.

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **transform** – The affine transform matrix for the given window

Return type Affine

write()
Write the arr array into indexed bands of the dataset.

If *indexes* is a list, the src must be a 3D array of matching shape. If an int, the src must be a 2D array.

See *read()* for usage of the optional *window* argument.

write_band()
Write the src array into the *bidx* band.

Band indexes begin with 1: read_band(1) returns the first band.

The optional *window* argument takes a tuple like:

((row_start, row_stop), (col_start, col_stop))

specifying a raster subset to write into.

write_colormap()
Write a colormap for a band to the dataset.

write_mask()
Write the valid data mask src array into the dataset's band mask.

The optional *window* argument takes a tuple like:

((row_start, row_stop), (col_start, col_stop))

specifying a raster subset to write into.

write_transform()

xy(*row*, *col*, *offset='center'*)
Returns the coordinates `(x, y)` of a pixel at *row* and *col*. The pixel's center is returned by default, but a corner can be returned by setting *offset* to one of *ul, ur, ll, lr*.

Parameters

- **row** (*int*) – Pixel row.
- **col** (*int*) – Pixel column.
- **offset** (*str, optional*) – Determines if the returned coordinates are for the center of the pixel or for a corner.

Returns `(x, y)`

Return type tuple

class rasterio.io.**MemoryFile**(*file_or_bytes=None*, *dirname=None*, *filename=None*, *ext='.tif'*)
Bases: *rasterio._io.MemoryFileBase*

A BytesIO-like object, backed by an in-memory file.

This allows formatted files to be read and written without I/O.

A MemoryFile created with initial bytes becomes immutable. A MemoryFile created without initial bytes may be written to using either file-like or dataset interfaces.

Examples

A GeoTIFF can be loaded in memory and accessed using the GeoTIFF format driver

```
>>> with open('tests/data/RGB.byte.tif', 'rb') as f, MemoryFile(f) as memfile:
...     with memfile.open() as src:
...         pprint.pprint(src.profile)
...
{'count': 3,
 'crs': CRS({'init': 'epsg:32618'}),
 'driver': 'GTiff',
 'dtype': 'uint8',
 'height': 718,
 'interleave': 'pixel',
 'nodata': 0.0,
 'tiled': False,
```

(continues on next page)

(continued from previous page)

```
 'transform': Affine(300.0379266750948, 0.0, 101985.0,
       0.0, -300.041782729805, 2826915.0),
 'width': 791}
```

close()

exists()
Test if the in-memory file exists.

Returns True if the in-memory file exists.

Return type bool

getbuffer()
Return a view on bytes of the file.

open(*driver=None*, *width=None*, *height=None*, *count=None*, *crs=None*, *transform=None*, *dtype=None*, *nodata=None*, *sharing=False*, ***kwargs*)
Open the file and return a Rasterio dataset object.

If data has already been written, the file is opened in 'r' mode. Otherwise, the file is opened in 'w' mode.

Parameters

- **well that there is no path parameter** (*Note*) –
- **a single dataset and there is no need to specify a** (*contains*) –
- **path.** –
- **parameters are optional and have the same semantics as the** (*Other*) –
- **of rasterio.open()** (*parameters*) –

read()
Read size bytes from MemoryFile.

seek()

tell()

write()
Write data bytes to MemoryFile

class rasterio.io.**ZipMemoryFile**(*file_or_bytes=None*)
Bases: *rasterio.io.MemoryFile*

A read-only BytesIO-like object backed by an in-memory zip file.

This allows a zip file containing formatted files to be read without I/O.

close()

exists()
Test if the in-memory file exists.

Returns True if the in-memory file exists.

Return type bool

getbuffer()
Return a view on bytes of the file.

open(*path*, *driver=None*, *sharing=False*, ***kwargs*)
Open a dataset within the zipped stream.

Parameters

- **path** (*str*) – Path to a dataset in the zip file, relative to the root of the archive.
- **parameters are optional and have the same semantics as the** (*Other*) –
- **of rasterio.open()** (*parameters*) –

Returns

Return type A Rasterio dataset object

read()
Read size bytes from MemoryFile.

seek()

tell()

write()
Write data bytes to MemoryFile

rasterio.io.**get_writer_for_driver**(*driver*)
Return the writer class appropriate for the specified driver.

rasterio.io.**get_writer_for_path**(*path*, *driver=None*)
Return the writer class appropriate for the existing dataset.

rasterio.mask module

Mask the area outside of the input shapes with no data.

rasterio.mask.**mask**(*dataset*, *shapes*, *all_touched=False*, *invert=False*, *nodata=None*, *filled=True*, *crop=False*, *pad=False*, *pad_width=0.5*, *indexes=None*)
Creates a masked or filled array using input shapes. Pixels are masked or set to nodata outside the input shapes, unless *invert* is *True*.

If shapes do not overlap the raster and crop=True, a ValueError is raised. Otherwise, a warning is raised.

Parameters

- **dataset** (*a dataset object opened in 'r' mode*) – Raster to which the mask will be applied.
- **shapes** (*iterable object*) – The values must be a GeoJSON-like dict or an object that implements the Python geo interface protocol (such as a Shapely Polygon).
- **all_touched** (*bool (opt)*) – Include a pixel in the mask if it touches any of the shapes. If False (default), include a pixel only if its center is within one of the shapes, or if it is selected by Bresenham's line algorithm.
- **invert** (*bool (opt)*) – If False (default) pixels outside shapes will be masked. If True, pixels inside shape will be masked.
- **nodata** (*int or float (opt)*) – Value representing nodata within each raster band. If not set, defaults to the nodata value for the input raster. If there is no set nodata value for the raster, it defaults to 0.

- **filled** (*bool (opt)*) – If True, the pixels outside the features will be set to nodata. If False, the output array will contain the original pixel data, and only the mask will be based on shapes. Defaults to True.
- **crop** (*bool (opt)*) – Whether to crop the raster to the extent of the shapes. Defaults to False.
- **pad** (*bool (opt)*) – If True, the features will be padded in each direction by one half of a pixel prior to cropping raster. Defaults to False.
- **indexes** (*list of ints or a single int (opt)*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.

Returns

Two elements:

masked [numpy ndarray or numpy.ma.MaskedArray] Data contained in the raster after applying the mask. If *filled* is *True* and *invert* is *False*, the return will be an array where pixels outside shapes are set to the nodata value (or nodata inside shapes if *invert* is *True*).

If *filled* is *False*, the return will be a MaskedArray in which pixels outside shapes are *True* (or *False* if *invert* is *True*).

out_transform [affine.Affine()] Information for mapping pixel coordinates in *masked* to another coordinate system.

Return type tuple

rasterio.mask.**raster_geometry_mask**(*dataset*, *shapes*, *all_touched=False*, *invert=False*, *crop=False*, *pad=False*, *pad_width=0.5*)

Create a mask from shapes, transform, and optional window within original raster.

By default, mask is intended for use as a numpy mask, where pixels that overlap shapes are False.

If shapes do not overlap the raster and crop=True, a ValueError is raised. Otherwise, a warning is raised, and a completely True mask is returned (if invert is False).

Parameters

- **dataset** (*a dataset object opened in 'r' mode*) – Raster for which the mask will be created.
- **shapes** (*iterable object*) – The values must be a GeoJSON-like dict or an object that implements the Python geo interface protocol (such as a Shapely Polygon).
- **all_touched** (*bool (opt)*) – Include a pixel in the mask if it touches any of the shapes. If False (default), include a pixel only if its center is within one of the shapes, or if it is selected by Bresenham's line algorithm.
- **invert** (*bool (opt)*) – If False (default), mask will be *False* inside shapes and *True* outside. If True, mask will be *True* inside shapes and *False* outside.
- **crop** (*bool (opt)*) – Whether to crop the dataset to the extent of the shapes. Defaults to False.
- **pad** (*bool (opt)*) – If True, the features will be padded in each direction by one half of a pixel prior to cropping dataset. Defaults to False.
- **pad_width** (*float (opt)*) – If pad is set (to maintain back-compatibility), then this will be the pixel-size width of the padding around the mask.

Returns

Three elements:

mask [numpy ndarray of type 'bool'] Mask that is *True* outside shapes, and *False* within shapes.

out_transform [affine.Affine()] Information for mapping pixel coordinates in *masked* to another coordinate system.

window: rasterio.windows.Window instance Window within original raster covered by shapes. None if crop is False.

Return type tuple

rasterio.merge module

Copy valid pixels from input files to an output file.

`rasterio.merge.`**`copy_first`**(*merged_data*, *new_data*, *merged_mask*, *new_mask*, ***kwargs*)

`rasterio.merge.`**`copy_last`**(*merged_data*, *new_data*, *merged_mask*, *new_mask*, ***kwargs*)

`rasterio.merge.`**`copy_max`**(*merged_data*, *new_data*, *merged_mask*, *new_mask*, ***kwargs*)

`rasterio.merge.`**`copy_min`**(*merged_data*, *new_data*, *merged_mask*, *new_mask*, ***kwargs*)

`rasterio.merge.`**`merge`**(*datasets*, *bounds=None*, *res=None*, *nodata=None*, *dtype=None*, *precision=None*, *indexes=None*, *output_count=None*, *resampling=<Resampling.nearest: 0>*, *method='first'*, *dst_path=None*, *dst_kwds=None*)

Copy valid pixels from input files to an output file.

All files must have the same number of bands, data type, and coordinate reference system.

Input files are merged in their listed order using the reverse painter's algorithm (default) or another method. If the output file exists, its values will be overwritten by input values.

Geospatial bounds and resolution of a new output file in the units of the input file coordinate reference system may be provided and are otherwise taken from the first input file.

Parameters

- **`datasets`** (*`list of dataset objects opened in 'r' mode, filenames or pathlib.Path objects`*) – source datasets to be merged.
- **`bounds`** (*`tuple, optional`*) – Bounds of the output image (left, bottom, right, top). If not set, bounds are determined from bounds of input rasters.
- **`res`** (*`tuple, optional`*) – Output resolution in units of coordinate reference system. If not set, the resolution of the first raster is used. If a single value is passed, output pixels will be square.
- **`nodata`** (*`float, optional`*) – nodata value to use in output file. If not set, uses the nodata value in the first input raster.
- **`dtype`** (*`numpy dtype or string`*) – dtype to use in outputfile. If not set, uses the dtype value in the first input raster.
- **`precision`** (*`float, optional`*) – Number of decimal points of precision when computing inverse transform.
- **`indexes`** (*`list of ints or a single int, optional`*) – bands to read and merge

- **output_count** (*int, optional*) – If using callable it may be useful to have additional bands in the output in addition to the indexes specified for read
- **resampling** (Resampling, *optional*) – Resampling algorithm used when reading input files. Default: *Resampling.nearest*.
- **method** (*str or callable*) –

  pre-defined method: first: reverse painting last: paint valid new on top of existing min: pixel-wise min of existing and new max: pixel-wise max of existing and new

  or custom callable with signature:

  def function(merged_data, new_data, merged_mask, new_mask, index=None, roff=None, coff=None):

  merged_data [array_like] array to update with new_data

  new_data [array_like] data to merge same shape as merged_data

  merged_mask, new_mask [array_like] boolean masks where merged/new data pixels are invalid same shape as merged_data

  index: int index of the current dataset within the merged dataset collection

  roff: int row offset in base array

  coff: int column offset in base array
- **dst_path** (*str or Pathlike, optional*) – Path of output dataset
- **dst_kwds** (*dict, optional*) – Dictionary of creation options and other paramters that will be overlaid on the profile of the output dataset.

Returns

Two elements:

dest: numpy ndarray Contents of all input rasters in single array

out_transform: affine.Affine() Information for mapping pixel coordinates in *dest* to another coordinate system

Return type tuple

rasterio.path module

Dataset paths, identifiers, and filenames

class rasterio.path.**ParsedPath**(*path*, *archive*, *scheme*)

Bases: *rasterio.path.Path*

Result of parsing a dataset URI/Path

path

Parsed path. Includes the hostname and query string in the case of a URI.

Type str

archive

Parsed archive path.

Type str

scheme

URI scheme such as "https" or "zip+s3".

Type str

archive

classmethod from_uri(*uri*)

property is_local
Test if the path is a local URI

property is_remote
Test if the path is a remote, network URI

property name
The parsed path's original URI

path

scheme

class rasterio.path.**Path**
Bases: object

Base class for dataset paths

as_vsi()

class rasterio.path.**UnparsedPath**(*path*)
Bases: *rasterio.path.Path*

Encapsulates legacy GDAL filenames

path
The legacy GDAL filename.

Type str

property name
The unparsed path's original path

path

rasterio.path.**parse_path**(*path*)
Parse a dataset's identifier or path into its parts

Parameters **path** (*str or path-like object*) – The path to be parsed.

Returns

Return type *ParsedPath* or *UnparsedPath*

Notes

When legacy GDAL filenames are encountered, they will be returned in a UnparsedPath.

rasterio.path.**vsi_path**(*path*)
Convert a parsed path to a GDAL VSI path

Parameters **path** (Path) – A ParsedPath or UnparsedPath object.

Returns

Return type str

rasterio.plot module

Implementations of various common operations.

Including *show()* for displaying an array or with matplotlib. Most can handle a numpy array or *rasterio.Band()*. Primarily supports *$ rio insp*.

rasterio.plot.**adjust_band**(*band*, *kind='linear'*)

Adjust a band to be between 0 and 1.

Parameters

- **band** (*array, shape (height, width)*) – A band of a raster object.
- **kind** (*'linear'*) – The kind of normalization to apply. For now, there is only one option ('linear').

Returns **band_normed** – An adjusted version of the input band.

Return type array, shape (height, width)

rasterio.plot.**get_plt**()

import matplotlib.pyplot raise import error if matplotlib is not installed

rasterio.plot.**plotting_extent**(*source*, *transform=None*)

Returns an extent in the format needed for matplotlib's imshow (left, right, bottom, top) instead of rasterio's bounds (left, bottom, right, top)

Parameters

- **source** (*array or dataset object opened in 'r' mode*) – If array, data in the order rows, columns and optionally bands. If array is band order (bands in the first dimension), use arr[0]
- **transform** (*Affine, required if source is array*) – Defines the affine transform if source is an array

Returns left, right, bottom, top

Return type tuple of float

rasterio.plot.**reshape_as_image**(*arr*)

Returns the source array reshaped into the order expected by image processing and visualization software (matplotlib, scikit-image, etc) by swapping the axes order from (bands, rows, columns) to (rows, columns, bands)

Parameters **arr** (*array-like of shape (bands, rows, columns)*) – image to reshape

rasterio.plot.**reshape_as_raster**(*arr*)

Returns the array in a raster order by swapping the axes order from (rows, columns, bands) to (bands, rows, columns)

Parameters **arr** (*array-like in the image form of (rows, columns, bands)*) – image to reshape

rasterio.plot.**show**(*source*, *with_bounds=True*, *contour=False*, *contour_label_kws=None*, *ax=None*, *title=None*, *transform=None*, *adjust='linear'*, ***kwargs*)

Display a raster or raster band using matplotlib.

Parameters

- **source** (*array or dataset object opened in 'r' mode or Band or tuple(dataset, bidx)*) – If Band or tuple (dataset, bidx), display the selected band. If raster dataset display the rgb image as defined in the colorinterp metadata, or default to first band.
- **with_bounds** (*bool (opt)*) – Whether to change the image extent to the spatial bounds of the image, rather than pixel coordinates. Only works when source is (raster dataset, bidx) or raster dataset.
- **contour** (*bool (opt)*) – Whether to plot the raster data as contours
- **contour_label_kws** (*dictionary (opt)*) – Keyword arguments for labeling the contours, empty dictionary for no labels.
- **ax** (*matplotlib axes (opt)*) – Axes to plot on, otherwise uses current axes.
- **title** (*str, optional*) – Title for the figure.
- **transform** (*Affine, optional*) – Defines the affine transform if source is an array
- **adjust** (*'linear' | None*) – If the plotted data is an RGB image, adjust the values of each band so that they fall between 0 and 1 before plotting. If 'linear', values will be adjusted by the min / max of each band. If None, no adjustment will be applied.
- ****kwargs** (*key, value pairings optional*) – These will be passed to the matplotlib imshow or contour method depending on contour argument. See full lists at: http://matplotlib.org/api/axes_api.html?highlight=imshow#matplotlib.axes.Axes.imshow or http://matplotlib.org/api/axes_api.html?highlight=imshow#matplotlib.axes.Axes.contour

Returns **ax** – Axes with plot.

Return type matplotlib Axes

rasterio.plot.**show_hist**(*source*, *bins=10*, *masked=True*, *title='Histogram'*, *ax=None*, *label=None*, ***kwargs*)

Easily display a histogram with matplotlib.

Parameters

- **source** (*array or dataset object opened in 'r' mode or Band or tuple(dataset, bidx)*) – Input data to display. The first three arrays in multi-dimensional arrays are plotted as red, green, and blue.
- **bins** (*int, optional*) – Compute histogram across N bins.
- **masked** (*bool, optional*) – When working with a *rasterio.Band()* object, specifies if the data should be masked on read.
- **title** (*str, optional*) – Title for the figure.
- **ax** (*matplotlib axes (opt)*) – The raster will be added to this axes if passed.
- **label** (*matplotlib labels (opt)*) – If passed, matplotlib will use this label list. Otherwise, a default label list will be automatically created
- ****kwargs** (*optional keyword arguments*) – These will be passed to the matplotlib hist method. See full list at: http://matplotlib.org/api/axes_api.html?highlight=imshow#matplotlib.axes.Axes.hist

rasterio.profiles module

Raster dataset profiles.

class rasterio.profiles.**DefaultGTiffProfile**(*data={}*, ***kwds*)
Bases: *rasterio.profiles.Profile*

Tiled, band-interleaved, LZW-compressed, 8-bit GTiff.

defaults = {'blockxsize': 256, 'blockysize': 256, 'compress': 'lzw', 'driver': 'GT

class rasterio.profiles.**Profile**(*data={}*, ***kwds*)
Bases: collections.UserDict

Base class for Rasterio dataset profiles.

Subclasses will declare driver-specific creation options.

defaults = {}

rasterio.sample module

rasterio.sample.**sample_gen**(*dataset*, *xy*, *indexes=None*, *masked=False*)
Sample pixels from a dataset

Parameters

- **dataset** (*rasterio Dataset*) – Opened in "r" mode.
- **xy** (*iterable*) – Pairs of x, y coordinates in the dataset's reference system.
- **indexes** (*int or list of int*) – Indexes of dataset bands to sample.
- **masked** (*bool, default: False*) – Whether to mask samples that fall outside the extent of the dataset.

Yields *array* – A array of length equal to the number of specified indexes containing the dataset values for the bands corresponding to those indexes.

rasterio.shutil module

Raster file management.

rasterio.shutil.**copy**()
Copy a raster from a path or open dataset handle to a new destination with driver specific creation options.

Parameters

- **src** (*str or pathlib.Path or dataset object opened in 'r' mode*) – Source dataset
- **dst** (*str or pathlib.Path*) – Output dataset path
- **driver** (*str, optional*) – Output driver name
- **strict** (*bool, optional. Default: True*) – Indicates if the output must be strictly equivalent or if the driver may adapt as necessary
- **creation_options** (**kwargs, optional) – Creation options for output dataset

Returns

Return type None

rasterio.shutil.**copyfiles**()
Copy files associated with a dataset from one location to another.

Parameters

- **src** (*str or pathlib.Path*) – Source dataset
- **dst** (*str or pathlib.Path*) – Target dataset

Returns

Return type None

rasterio.shutil.**delete**()
Delete a GDAL dataset

Parameters

- **path** (*path*) – Path to dataset to delete
- **driver** (*str or None, optional*) – Name of driver to use for deleting. Defaults to whatever GDAL determines is the appropriate driver

rasterio.shutil.**exists**()
Determine if a dataset exists by attempting to open it.

Parameters **path** (*str*) – Path to dataset

rasterio.tools module

Rasterio tools module

See this RFC about Rasterio tools: https://github.com/mapbox/rasterio/issues/1300.

class rasterio.tools.**JSONSequenceTool**(*func*)
Bases: object

Extracts data from a dataset file and saves a JSON sequence

rasterio.transform module

Geospatial transforms

class rasterio.transform.**TransformMethodsMixin**
Bases: object

Mixin providing methods for calculations related to transforming between rows and columns of the raster array and the coordinates.

These methods are wrappers for the functionality in *rasterio.transform* module.

A subclass with this mixin MUST provide a *transform* property.

index(*x*, *y*, *op=<built-in function floor>*, *precision=None*)
Returns the (row, col) index of the pixel containing (x, y) given a coordinate reference system.

Use an epsilon, magnitude determined by the precision parameter and sign determined by the op function:

positive for floor, negative for ceil.

Parameters

- **x** (*float*) – x value in coordinate reference system

- **y** (*float*) – y value in coordinate reference system
- **op** (*function, optional (default: math.floor)*) – Function to convert fractional pixels to whole numbers (floor, ceiling, round)
- **precision** (*int, optional (default: None)*) – Decimal places of precision in indexing, as in *round()*.

Returns (row index, col index)

Return type tuple

xy(*row*, *col*, *offset='center'*)

Returns the coordinates `(x, y)` of a pixel at *row* and *col*. The pixel's center is returned by default, but a corner can be returned by setting *offset* to one of *ul, ur, ll, lr*.

Parameters

- **row** (*int*) – Pixel row.
- **col** (*int*) – Pixel column.
- **offset** (*str, optional*) – Determines if the returned coordinates are for the center of the pixel or for a corner.

Returns `(x, y)`

Return type tuple

rasterio.transform.**array_bounds**(*height*, *width*, *transform*)

Return the bounds of an array given height, width, and a transform.

Return the *west, south, east, north* bounds of an array given its height, width, and an affine transform.

rasterio.transform.**from_bounds**(*west*, *south*, *east*, *north*, *width*, *height*)

Return an Affine transformation given bounds, width and height.

Return an Affine transformation for a georeferenced raster given its bounds *west*, *south*, *east*, *north* and its *width* and *height* in number of pixels.

rasterio.transform.**from_gcps**(*gcps*)

Make an Affine transform from ground control points.

Parameters **gcps** (*sequence of GroundControlPoint*) – Such as the first item of a dataset's *gcps* property.

Returns

Return type Affine

rasterio.transform.**from_origin**(*west*, *north*, *xsize*, *ysize*)

Return an Affine transformation given upper left and pixel sizes.

Return an Affine transformation for a georeferenced raster given the coordinates of its upper left corner *west*, *north* and pixel sizes *xsize*, *ysize*.

rasterio.transform.**guard_transform**(*transform*)

Return an Affine transformation instance.

rasterio.transform.**rowcol**(*transform*, *xs*, *ys*, *op=<built-in function floor>*, *precision=None*)

Returns the rows and cols of the pixels containing (x, y) given a coordinate reference system.

Use an epsilon, magnitude determined by the precision parameter and sign determined by the op function:

positive for floor, negative for ceil.

Parameters

- **transform** (*Affine*) – Coefficients mapping pixel coordinates to coordinate reference system.
- **xs** (*list or float*) – x values in coordinate reference system
- **ys** (*list or float*) – y values in coordinate reference system
- **op** (*function*) – Function to convert fractional pixels to whole numbers (floor, ceiling, round)
- **precision** (*int or float, optional*) – An integer number of decimal points of precision when computing inverse transform, or an absolute float precision.

Returns

- **rows** (*list of ints*) – list of row indices
- **cols** (*list of ints*) – list of column indices

rasterio.transform.**tastes_like_gdal**(*seq*)
Return True if *seq* matches the GDAL geotransform pattern.

rasterio.transform.**xy**(*transform*, *rows*, *cols*, *offset='center'*)
Returns the x and y coordinates of pixels at *rows* and *cols*. The pixel's center is returned by default, but a corner can be returned by setting *offset* to one of *ul, ur, ll, lr*.

Parameters

- **transform** (*affine.Affine*) – Transformation from pixel coordinates to coordinate reference system.
- **rows** (*list or int*) – Pixel rows.
- **cols** (*list or int*) – Pixel columns.
- **offset** (*str, optional*) – Determines if the returned coordinates are for the center of the pixel or for a corner.

Returns

- **xs** (*list*) – x coordinates in coordinate reference system
- **ys** (*list*) – y coordinates in coordinate reference system

rasterio.vrt module

rasterio.vrt: a module concerned with GDAL VRTs

class rasterio.vrt.**WarpedVRT**
Bases: *rasterio._warp.WarpedVRTReaderBase*, *rasterio.windows.WindowMethodsMixin*, *rasterio.transform.TransformMethodsMixin*

A virtual warped dataset.

Abstracts the details of raster warping and allows access to data that is reprojected when read.

This class is backed by an in-memory GDAL VRTWarpedDataset VRT file.

Parameters

- **src_dataset** (*dataset object*) – The warp source.

- **src_crs** (*CRS or str, optional*) – Overrides the coordinate reference system of *src_dataset*.
- **src_transfrom** (*Affine, optional*) – Overrides the transform of *src_dataset*.
- **src_nodata** (*float, optional*) – Overrides the nodata value of *src_dataset*, which is the default.
- **crs** (*CRS or str, optional*) – The coordinate reference system at the end of the warp operation. Default: the crs of *src_dataset*. dst_crs is a deprecated alias for this parameter.
- **transform** (*Affine, optional*) – The transform for the virtual dataset. Default: will be computed from the attributes of *src_dataset*. dst_transform is a deprecated alias for this parameter.
- **height** (*int, optional*) – The dimensions of the virtual dataset. Defaults: will be computed from the attributes of *src_dataset*. dst_height and dst_width are deprecated alias for these parameters.
- **width** (*int, optional*) – The dimensions of the virtual dataset. Defaults: will be computed from the attributes of *src_dataset*. dst_height and dst_width are deprecated alias for these parameters.
- **nodata** (*float, optional*) – Nodata value for the virtual dataset. Default: the nodata value of *src_dataset* or 0.0. dst_nodata is a deprecated alias for this parameter.
- **resampling** (*Resampling, optional*) – Warp resampling algorithm. Default: *Resampling.nearest*.
- **tolerance** (*float, optional*) – The maximum error tolerance in input pixels when approximating the warp transformation. Default: 0.125, or one-eigth of a pixel.
- **src_alpha** (*int, optional*) – Index of a source band to use as an alpha band for warping.
- **add_alpha** (*bool, optional*) – Whether to add an alpha masking band to the virtual dataset. Default: False. This option will cause deletion of the VRT nodata value.
- **init_dest_nodata** (*bool, optional*) – Whether or not to initialize output to *nodata*. Default: True.
- **warp_mem_limit** (*int, optional*) – The warp operation's memory limit in MB. The default (0) means 64 MB with GDAL 2.2.
- **dtype** (*str, optional*) – The working data type for warp operation and output.
- **warp_extras** (*dict*) – GDAL extra warp options. See https://gdal.org/doxygen/structGDALWarpOptions.html.

src_dataset
The dataset object to be virtually warped.

Type dataset

resampling
One of the values from rasterio.enums.Resampling. The default is *Resampling.nearest*.

Type int

tolerance
The maximum error tolerance in input pixels when approximating the warp transformation. The default is 0.125.

Type float

src_nodata
The source nodata value. Pixels with this value will not be used for interpolation. If not set, it will be default to the nodata value of the source image, if available.

Type int or float, optional

dst_nodata
The nodata value used to initialize the destination; it will remain in all areas not covered by the reprojected source. Defaults to the value of src_nodata, or 0 (gdal default).

Type int or float, optional

working_dtype
The working data type for warp operation and output.

Type str, optional

warp_extras
GDAL extra warp options. See https://gdal.org/doxygen/structGDALWarpOptions.html.

Type dict

Examples

```
>>> with rasterio.open('tests/data/RGB.byte.tif') as src:
...     with WarpedVRT(src, crs='EPSG:3857') as vrt:
...         data = vrt.read()
```

block_shapes
An ordered list of block shapes for each bands

Shapes are tuples and have the same ordering as the dataset's shape: (count of image rows, count of image columns).

Returns

Return type list

block_size()
Returns the size in bytes of a particular block

Only useful for TIFF formatted datasets.

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.
- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type int

block_window()
Returns the window for a particular block

Parameters

- **bidx** (*int*) – Band index, starting with 1.
- **i** (*int*) – Row index of the block, starting with 0.

- **j** (*int*) – Column index of the block, starting with 0.

Returns

Return type *Window*

block_windows()

Iterator over a band's blocks and their windows

The primary use of this method is to obtain windows to pass to *read()* for highly efficient access to raster block data.

The positional parameter *bidx* takes the index (starting at 1) of the desired band. This iterator yields blocks "left to right" and "top to bottom" and is similar to Python's `enumerate()` in that the first element is the block index and the second is the dataset window.

Blocks are built-in to a dataset and describe how pixels are grouped within each band and provide a mechanism for efficient I/O. A window is a range of pixels within a single band defined by row start, row stop, column start, and column stop. For example, `((0, 2), (0, 2))` defines a `2 x 2` window at the upper left corner of a raster band. Blocks are referenced by an `(i, j)` tuple where `(0, 0)` would be a band's upper left block.

Raster I/O is performed at the block level, so accessing a window spanning multiple rows in a striped raster requires reading each row. Accessing a `2 x 2` window at the center of a `1800 x 3600` image requires reading 2 rows, or 7200 pixels just to get the target 4. The same image with internal `256 x 256` blocks would require reading at least 1 block (if the window entire window falls within a single block) and at most 4 blocks, or at least 512 pixels and at most 2048.

Given an image that is `512 x 512` with blocks that are `256 x 256`, its blocks and windows would look like:

```
Blocks:

        0       256     512
      0 +--------+--------+
        |        |        |
        | (0, 0) | (0, 1) |
        |        |        |
    256 +--------+--------+
        |        |        |
        | (1, 0) | (1, 1) |
        |        |        |
    512 +--------+--------+


Windows:

    UL: ((0, 256), (0, 256))
    UR: ((0, 256), (256, 512))
    LL: ((256, 512), (0, 256))
    LR: ((256, 512), (256, 512))
```

Parameters **bidx** (*int, optional*) – The band index (using 1-based indexing) from which to extract windows. A value less than 1 uses the first band if all bands have homogeneous windows and raises an exception otherwise.

Yields *block, window*

bounds

Returns the lower left and upper right bounds of the dataset in the units of its coordinate reference system.

The returned value is a tuple: (lower left x, lower left y, upper right x, upper right y)

checksum()
Compute an integer checksum for the stored band

Parameters

- **bidx** (*int*) – The band's index (1-indexed).
- **window** (*tuple, optional*) – A window of the band. Default is the entire extent of the band.

Returns

Return type An int.

close()
Close the dataset

closed
Test if the dataset is closed

Returns

Return type bool

colorinterp
Returns a sequence of `ColorInterp.<enum>` representing color interpretation in band order.

To set color interpretation, provide a sequence of `ColorInterp.<enum>`:

import rasterio from rasterio.enums import ColorInterp

with rasterio.open('rgba.tif', 'r+') as src:

src.colorinterp = (ColorInterp.red, ColorInterp.green, ColorInterp.blue, ColorInterp.alpha)

Returns

Return type tuple

colormap()
Returns a dict containing the colormap for a band or None.

compression

count
The number of raster bands in the dataset

Returns

Return type int

crs
The dataset's coordinate reference system

dataset_mask()
Get the dataset's 2D valid data mask.

Parameters

- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array with the same dimensions and shape into which data will be placed.

Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

Cannot be combined with *out_shape*.

- **out_shape** (*tuple, optional*) – A tuple describing the output array's shape. Allows for decimated reads without constructing an output Numpy array.

  Cannot be combined with *out*.

- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.
- **boundless** (bool, optional (default *False*)) – If *True*, windows that extend beyond the dataset's extent are permitted and partially or completely filled arrays will be returned as appropriate.
- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

Returns The dtype of this array is uint8. 0 = nodata, 255 = valid data.

Return type Numpy ndarray or a view on a Numpy ndarray

Notes

Note: as with Numpy ufuncs, an object is returned even if you use the optional *out* argument and the return value shall be preferentially used by callers.

The dataset mask is calculated based on the individual band masks according to the following logic, in order of precedence:

1. If a .msk file, dataset-wide alpha, or internal mask exists it will be used for the dataset mask.
2. Else if the dataset is a 4-band 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 255.

Note that this differs from read_masks and GDAL RFC15 in that it applies per-dataset, not per-band (see https://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask)

descriptions

Descriptions for each dataset band

To set descriptions, one for each band is required.

Returns

Return type list of str

driver

dtypes

The data types of each band in index order

Returns

Return type list of str

files
Returns a sequence of files associated with the dataset.

Returns

Return type tuple

gcps
ground control points and their coordinate reference system.

This property is a 2-tuple, or pair: (gcps, crs).

gcps [list of GroundControlPoint] Zero or more ground control points.

crs: CRS The coordinate reference system of the ground control points.

get_gcps()
Get GCPs and their associated CRS.

get_nodatavals()

get_tag_item()
Returns tag item value

Parameters

- **ns** (*str*) – The key for the metadata item to fetch.
- **dm** (*str*) – The domain to fetch for.
- **bidx** (*int*) – Band index, starting with 1.
- **ovr** (*int*) – Overview level

Returns

Return type str

get_transform()
Returns a GDAL geotransform in its native form.

height

index(*x*, *y*, *op=<built-in function floor>*, *precision=None*)
Returns the (row, col) index of the pixel containing (x, y) given a coordinate reference system.

Use an epsilon, magnitude determined by the precision parameter and sign determined by the op function:

positive for floor, negative for ceil.

Parameters

- **x** (*float*) – x value in coordinate reference system
- **y** (*float*) – y value in coordinate reference system
- **op** (*function, optional (default: math.floor)*) – Function to convert fractional pixels to whole numbers (floor, ceiling, round)
- **precision** (*int, optional (default: None)*) – Decimal places of precision in indexing, as in *round()*.

Returns (row index, col index)

Return type tuple

indexes
The 1-based indexes of each band in the dataset

For a 3-band dataset, this property will be `[1, 2, 3]`.

Returns

Return type list of int

interleaving

is_tiled

lnglat()

mask_flag_enums
Sets of flags describing the sources of band masks.

all_valid: There are no invalid pixels, all mask values will be

255. When used this will normally be the only flag set.

per_dataset: The mask band is shared between all bands on the dataset.

alpha: The mask band is actually an alpha band and may have values other than 0 and 255.

nodata: Indicates the mask is actually being generated from nodata values (mutually exclusive of "alpha").

Returns One list of rasterio.enums.MaskFlags members per band.

Return type list [, list*]

Examples

For a 3 band dataset that has masks derived from nodata values:

```
>>> dataset.mask_flag_enums
([<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>], [<MaskFlags.nodata: 8>])
>>> band1_flags = dataset.mask_flag_enums[0]
>>> rasterio.enums.MaskFlags.nodata in band1_flags
True
>>> rasterio.enums.MaskFlags.alpha in band1_flags
False
```

meta
The basic metadata of this dataset.

mode

name

nodata
The dataset's single nodata value

Notes

May be set.

Returns

Return type float

nodatavals
Nodata values for each band

Notes

This may not be set.

Returns

Return type list of float

offsets
Raster offset for each dataset band

To set offsets, one for each band is required.

Returns

Return type list of float

options

overviews()

photometric

profile
Basic metadata and creation options of this dataset.

May be passed as keyword arguments to *rasterio.open()* to create a clone of this dataset.

read()
Read a dataset's raw pixels as an N-d array

This data is read from the dataset's band cache, which means that repeated reads of the same windows may avoid I/O.

Parameters

- **indexes** (*list of ints or a single int, optional*) – If *indexes* is a list, the result is a 3D array, but is a 2D array if it is a band index number.
- **out** (*numpy ndarray, optional*) – As with Numpy ufuncs, this is an optional reference to an output array into which data will be placed. If the height and width of *out* differ from that of the specified window (see below), the raster image will be decimated or replicated using the specified resampling method (also see below).

  Note: the method's return value may be a view on this array. In other words, *out* is likely to be an incomplete representation of the method's results.

  This parameter cannot be combined with *out_shape*.
- **out_dtype** (*str or numpy dtype*) – The desired output data type. For example: 'uint8' or rasterio.uint16.

- **out_shape** (*tuple, optional*) – A tuple describing the shape of a new output array. See *out* (above) for notes on image decimation and replication.

  Cannot combined with *out*.

- **window** (*a pair (tuple) of pairs of ints or Window, optional*) – The optional *window* argument is a 2 item tuple. The first item is a tuple containing the indexes of the rows at which the window starts and stops and the second is a tuple containing the indexes of the columns at which the window starts and stops. For example, ((0, 2), (0, 2)) defines a 2x2 window at the upper left of the raster dataset.

- **masked** (*bool, optional*) – If *masked* is *True* the return value will be a masked array. Otherwise (the default) the return value will be a regular array. Masks will be exactly the inverse of the GDAL RFC 15 conforming arrays returned by read_masks().

- **resampling** (Resampling) – By default, pixel values are read raw or interpolated using a nearest neighbor algorithm from the band cache. Other resampling algorithms may be specified. Resampled pixels are not cached.

- **fill_value** (*scalar*) – Fill value applied in the *boundless=True* case only.

- **kwargs** (*dict*) – This is only for backwards compatibility. No keyword arguments are supported other than the ones named above.

Returns

- *Numpy ndarray or a view on a Numpy ndarray*
- **Note** (*as with Numpy ufuncs, an object is returned even if you*)
- use the optional *out* argument and the return value shall be
- *preferentially used by callers.*

read_crs()
Return the GDAL dataset's stored CRS

read_masks()
Read raster band masks as a multidimensional array

read_transform()
Return the stored GDAL GeoTransform

res
Returns the (width, height) of pixels in the units of its coordinate reference system.

rpcs
Rational polynomial coefficients mapping between pixel and geodetic coordinates.

This property is a dict-like object.

rpcs : RPC instance containing coefficients. Empty if dataset does not have any metadata in the "RPC" domain.

sample()
Get the values of a dataset at certain positions

Values are from the nearest pixel. They are not interpolated.

Parameters

- **xy** (*iterable*) – Pairs of x, y coordinates (floats) in the dataset's reference system.
- **indexes** (*int or list of int*) – Indexes of dataset bands to sample.

- **masked** (*bool, default: False*) – Whether to mask samples that fall outside the extent of the dataset.

Returns Arrays of length equal to the number of specified indexes containing the dataset values for the bands corresponding to those indexes.

Return type iterable

scales
Raster scale for each dataset band

To set scales, one for each band is required.

Returns

Return type list of float

shape

start()
Start the dataset's life cycle

stop()
Close the GDAL dataset handle

subdatasets
Sequence of subdatasets

tag_namespaces()
Get a list of the dataset's metadata domains.

Returned items may be passed as *ns* to the tags method.

Parameters

- **int** (*bidx*) – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.
- **optional** – Can be used to select a specific band, otherwise the dataset's general metadata domains are returned.

Returns

Return type list of str

tags()
Returns a dict containing copies of the dataset or band's tags.

Tags are pairs of key and value strings. Tags belong to namespaces. The standard namespaces are: default (None) and 'IMAGE_STRUCTURE'. Applications can create their own additional namespaces.

The optional bidx argument can be used to select the tags of a specific band. The optional ns argument can be used to select a namespace other than the default.

transform
The dataset's georeferencing transformation matrix

This transform maps pixel row/column coordinates to coordinates in the dataset's coordinate reference system.

Returns

Return type Affine

units
one units string for each dataset band

Possible values include 'meters' or 'degC'. See the Pint project for a suggested list of units.

To set units, one for each band is required.

Returns

Return type list of str

Type A list of str

width

window(*left*, *bottom*, *right*, *top*, *precision=None*)
Get the window corresponding to the bounding coordinates.

The resulting window is not cropped to the row and column limits of the dataset.

Parameters

- **left** (*float*) – Left (west) bounding coordinate
- **bottom** (*float*) – Bottom (south) bounding coordinate
- **right** (*float*) – Right (east) bounding coordinate
- **top** (*float*) – Top (north) bounding coordinate
- **precision** (*int, optional*) – Number of decimal points of precision when computing inverse transform.

Returns **window**

Return type *Window*

window_bounds(*window*)
Get the bounds of a window

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **bounds** – x_min, y_min, x_max, y_max for the given window

Return type tuple

window_transform(*window*)
Get the affine transform for a dataset window.

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **transform** – The affine transform matrix for the given window

Return type Affine

write_transform()

xy(*row*, *col*, *offset='center'*)
Returns the coordinates (x, y) of a pixel at *row* and *col*. The pixel's center is returned by default, but a corner can be returned by setting *offset* to one of *ul, ur, ll, lr*.

Parameters

- **row** (*int*) – Pixel row.
- **col** (*int*) – Pixel column.
- **offset** (*str, optional*) – Determines if the returned coordinates are for the center of the pixel or for a corner.

Returns `(x, y)`

Return type tuple

rasterio.warp module

Raster warping and reprojection.

`rasterio.warp.`**`aligned_target`**(*transform*, *width*, *height*, *resolution*)

Aligns target to specified resolution

Parameters

- **`transform`** (*`Affine`*) – Input affine transformation matrix
- **`width`** (*`int`*) – Input dimensions
- **`height`** (*`int`*) – Input dimensions
- **`resolution`** (*`tuple (x resolution, y resolution) or float`*) – Target resolution, in units of target coordinate reference system.

Returns

- **transform** (*Affine*) – Output affine transformation matrix
- **width, height** (*int*) – Output dimensions

`rasterio.warp.`**`calculate_default_transform`**(*src_crs*, *dst_crs*, *width*, *height*, *left=None*, *bottom=None*, *right=None*, *top=None*, *gcps=None*, *rpcs=None*, *resolution=None*, *dst_width=None*, *dst_height=None*, ***kwargs*)

Output dimensions and transform for a reprojection.

Source and destination coordinate reference systems and output width and height are the first four, required, parameters. Source georeferencing can be specified using either ground control points (gcps) or spatial bounds (left, bottom, right, top). These two forms of georeferencing are mutually exclusive.

The destination transform is anchored at the left, top coordinate.

Destination width and height (and resolution if not provided), are calculated using GDAL's method for suggest warp output.

Parameters

- **`src_crs`** (*`CRS or dict`*) – Source coordinate reference system, in rasterio dict format. Example: CRS({'init': 'EPSG:4326'})
- **`dst_crs`** (*`CRS or dict`*) – Target coordinate reference system.
- **`width`** (*`int`*) – Source raster width and height.
- **`height`** (*`int`*) – Source raster width and height.
- **`left`** (*`float, optional`*) – Bounding coordinates in src_crs, from the bounds property of a raster. Required unless using gcps.
- **`bottom`** (*`float, optional`*) – Bounding coordinates in src_crs, from the bounds property of a raster. Required unless using gcps.
- **`right`** (*`float, optional`*) – Bounding coordinates in src_crs, from the bounds property of a raster. Required unless using gcps.
- **`top`** (*`float, optional`*) – Bounding coordinates in src_crs, from the bounds property of a raster. Required unless using gcps.

- **gcps** (*sequence of GroundControlPoint, optional*) – Instead of a bounding box for the source, a sequence of ground control points may be provided.
- **rpcs** (*RPC or dict, optional*) – Instead of a bounding box for the source, rational polynomial coefficients may be provided.
- **resolution** (*tuple (x resolution, y resolution) or float, optional*) – Target resolution, in units of target coordinate reference system.
- **dst_width** (*int, optional*) – Output file size in pixels and lines. Cannot be used together with resolution.
- **dst_height** (*int, optional*) – Output file size in pixels and lines. Cannot be used together with resolution.
- **kwargs** (*dict, optional*) – Additional arguments passed to transformation function.

Returns

- **transform** (*Affine*) – Output affine transformation matrix
- **width, height** (*int*) – Output dimensions

Notes

Some behavior of this function is determined by the CHECK_WITH_INVERT_PROJ environment variable:

YES: constrain output raster to extents that can be inverted avoids visual artifacts and coordinate discontinuties.

NO: reproject coordinates beyond valid bound limits

rasterio.warp.**reproject**(*source*, *destination=None*, *src_transform=None*, *gcps=None*, *rpcs=None*, *src_crs=None*, *src_nodata=None*, *dst_transform=None*, *dst_crs=None*, *dst_nodata=None*, *dst_resolution=None*, *src_alpha=0*, *dst_alpha=0*, *resampling=<Resampling.nearest: 0>*, *num_threads=1*, *init_dest_nodata=True*, *warp_mem_limit=0*, ***kwargs*)

Reproject a source raster to a destination raster.

If the source and destination are ndarrays, coordinate reference system definitions and affine transformation parameters or ground control points (gcps) are required for reprojection.

If the source and destination are rasterio Bands, shorthand for bands of datasets on disk, the coordinate reference systems and transforms or GCPs will be read from the appropriate datasets.

Parameters

- **source** (*ndarray or Band*) – The source is a 2 or 3-D ndarray, or a single or a multiple Rasterio Band object. The dimensionality of source and destination must match, i.e., for multiband reprojection the lengths of the first axes of the source and destination must be the same.
- **destination** (*ndarray or Band, optional*) – The destination is a 2 or 3-D ndarray, or a single or a multiple Rasterio Band object. The dimensionality of source and destination must match, i.e., for multiband reprojection the lengths of the first axes of the source and destination must be the same.
- **src_transform** (*affine.Affine(), optional*) – Source affine transformation. Required if source and destination are ndarrays. Will be derived from source if it is a rasterio Band. An error will be raised if this parameter is defined together with gcps.

- **gcps** (*sequence of GroundControlPoint, optional*) – Ground control points for the source. An error will be raised if this parameter is defined together with src_transform or rpcs.
- **rpcs** (*RPC or dict, optional*) – Rational polynomial coefficients for the source. An error will be raised if this parameter is defined together with src_transform or gcps.
- **src_crs** (CRS *or dict, optional*) – Source coordinate reference system, in rasterio dict format. Required if source and destination are ndarrays. Will be derived from source if it is a rasterio Band. Example: CRS({'init': 'EPSG:4326'})
- **src_nodata** (*int or float, optional*) – The source nodata value. Pixels with this value will not be used for interpolation. If not set, it will default to the nodata value of the source image if a masked ndarray or rasterio band, if available.
- **dst_transform** (*affine.Affine(), optional*) – Target affine transformation. Required if source and destination are ndarrays. Will be derived from target if it is a rasterio Band.
- **dst_crs** (CRS *or dict, optional*) – Target coordinate reference system. Required if source and destination are ndarrays. Will be derived from target if it is a rasterio Band.
- **dst_nodata** (*int or float, optional*) – The nodata value used to initialize the destination; it will remain in all areas not covered by the reprojected source. Defaults to the nodata value of the destination image (if set), the value of src_nodata, or 0 (GDAL default).
- **dst_resolution** (*tuple (x resolution, y resolution) or float, optional*) – Target resolution, in units of target coordinate reference system.
- **src_alpha** (*int, optional*) – Index of a band to use as the alpha band when warping.
- **dst_alpha** (*int, optional*) – Index of a band to use as the alpha band when warping.
- **resampling** (*int,* rasterio.enums.Resampling) – Resampling method to use. Default is *rasterio.enums.Resampling.nearest*. An exception will be raised for a method not supported by the running version of GDAL.
- **num_threads** (*int, optional*) – The number of warp worker threads. Default: 1.
- **init_dest_nodata** (*bool*) – Flag to specify initialization of nodata in destination; prevents overwrite of previous warps. Defaults to True.
- **warp_mem_limit** (*int, optional*) – The warp operation memory limit in MB. Larger values allow the warp operation to be carried out in fewer chunks. The amount of memory required to warp a 3-band uint8 2000 row x 2000 col raster to a destination of the same size is approximately 56 MB. The default (0) means 64 MB with GDAL 2.2.
- **kwargs** (*dict, optional*) – Additional arguments passed to transformation function.

Returns

- **destination** (*ndarray or Band*) – The transformed narray or Band.
- **dst_transform** (*Affine*) – THe affine transformation matrix of the destination.

rasterio.warp.**transform**(*src_crs*, *dst_crs*, *xs*, *ys*, *zs=None*)

Transform vectors from source to target coordinate reference system.

Transform vectors of x, y and optionally z from source coordinate reference system into target.

Parameters

- **src_crs** (*CRS or dict*) – Source coordinate reference system, as a rasterio CRS object. Example: CRS({'init': 'EPSG:4326'})
- **dst_crs** (*CRS or dict*) – Target coordinate reference system.
- **xs** (*array_like*) – Contains x values. Will be cast to double floating point values.
- **ys** (*array_like*) – Contains y values.
- **zs** (*array_like, optional*) – Contains z values. Assumed to be all 0 if absent.

Returns **out** – Tuple of x, y, and optionally z vectors, transformed into the target coordinate reference system.

Return type tuple of array_like, (xs, ys, [zs])

rasterio.warp.**transform_bounds**(*src_crs*, *dst_crs*, *left*, *bottom*, *right*, *top*, *densify_pts=21*)

Transform bounds from src_crs to dst_crs.

Optionally densifying the edges (to account for nonlinear transformations along these edges) and extracting the outermost bounds.

Note: this does not account for the antimeridian.

Parameters

- **src_crs** (*CRS or dict*) – Source coordinate reference system, in rasterio dict format. Example: CRS({'init': 'EPSG:4326'})
- **dst_crs** (*CRS or dict*) – Target coordinate reference system.
- **left** (*float*) – Bounding coordinates in src_crs, from the bounds property of a raster.
- **bottom** (*float*) – Bounding coordinates in src_crs, from the bounds property of a raster.
- **right** (*float*) – Bounding coordinates in src_crs, from the bounds property of a raster.
- **top** (*float*) – Bounding coordinates in src_crs, from the bounds property of a raster.
- **densify_pts** (*uint, optional*) – Number of points to add to each edge to account for nonlinear edges produced by the transform process. Large numbers will produce worse performance. Default: 21 (gdal default).

Returns **left, bottom, right, top** – Outermost coordinates in target coordinate reference system.

Return type float

rasterio.warp.**transform_geom**(*src_crs*, *dst_crs*, *geom*, *antimeridian_cutting=True*, *antimeridian_offset=10.0*, *precision=- 1*)

Transform geometry from source coordinate reference system into target.

Parameters

- **src_crs** (*CRS or dict*) – Source coordinate reference system, in rasterio dict format. Example: CRS({'init': 'EPSG:4326'})
- **dst_crs** (*CRS or dict*) – Target coordinate reference system.
- **geom** (*GeoJSON like dict object or iterable of GeoJSON like objects.*) –
- **antimeridian_cutting** (*bool, optional*) – If True, cut geometries at the antimeridian, otherwise geometries will not be cut (default). If False and GDAL is 2.2.0 or newer an exception is raised. Antimeridian cutting is always on as of GDAL 2.2.0 but this could produce an unexpected geometry.

- **antimeridian_offset** (*float*) – Offset from the antimeridian in degrees (default: 10) within which any geometries will be split.
- **precision** (*float*) – If >= 0, geometry coordinates will be rounded to this number of decimal places after the transform operation, otherwise original coordinate values will be preserved (default).

Returns **out** – Transformed geometry(s) in GeoJSON dict format

Return type GeoJSON like dict object or list of GeoJSON like objects.

rasterio.windows module

Window utilities and related functions.

A window is an instance of Window

Window(column_offset, row_offset, width, height)

or a 2D N-D array indexer in the form of a tuple.

((row_start, row_stop), (col_start, col_stop))

The latter can be evaluated within the context of a given height and width and a boolean flag specifying whether the evaluation is boundless or not. If boundless=True, negative index values do not mean index from the end of the array dimension as they do in the boundless=False case.

The newer float precision read-write window capabilities of Rasterio require instances of Window to be used.

class rasterio.windows.**Window**(*col_off*, *row_off*, *width*, *height*)

Bases: object

Windows are rectangular subsets of rasters.

This class abstracts the 2-tuples mentioned in the module docstring and adds methods and new constructors.

col_off, row_off

The offset for the window.

Type float

width, height

Lengths of the window.

Type float

Previously the lengths were called 'num_cols' and 'num_rows' but this is a bit confusing in the new float precision world and the attributes have been changed. The originals are deprecated.

col_off

crop(*height*, *width*)

Return a copy cropped to height and width

flatten()

A flattened form of the window.

Returns **col_off, row_off, width, height** – Window offsets and lengths.

Return type float

classmethod from_slices(*rows*, *cols*, *height=- 1*, *width=- 1*, *boundless=False*)
Construct a Window from row and column slices or tuples / lists of start and stop indexes. Converts the rows and cols to offsets, height, and width.

In general, indexes are defined relative to the upper left corner of the dataset: rows=(0, 10), cols=(0, 4) defines a window that is 4 columns wide and 10 rows high starting from the upper left.

Start indexes may be *None* and will default to 0. Stop indexes may be *None* and will default to width or height, which must be provided in this case.

Negative start indexes are evaluated relative to the lower right of the dataset: rows=(-2, None), cols=(-2, None) defines a window that is 2 rows high and 2 columns wide starting from the bottom right.

Parameters

- **rows** (*slice, tuple, or list*) – Slices or 2 element tuples/lists containing start, stop indexes.
- **cols** (*slice, tuple, or list*) – Slices or 2 element tuples/lists containing start, stop indexes.
- **height** (*float*) – A shape to resolve relative values against. Only used when a start or stop index is negative or a stop index is None.
- **width** (*float*) – A shape to resolve relative values against. Only used when a start or stop index is negative or a stop index is None.
- **boundless** (*bool, optional*) – Whether the inputs are bounded (default) or not.

Returns

Return type *Window*

height

intersection(*other*)
Return the intersection of this window and another

Parameters **other** (Window) – Another window

Returns

Return type *Window*

round_lengths(*op='floor'*, *pixel_precision=None*)
Return a copy with width and height rounded.

Lengths are rounded to the preceding (floor) or succeeding (ceil) whole number. The offsets are not changed.

Parameters

- **op** (*str*) – 'ceil' or 'floor'
- **pixel_precision** (*int, optional (default: None)*) – Number of places of rounding precision.

Returns

Return type *Window*

round_offsets(*op='floor'*, *pixel_precision=None*)
Return a copy with column and row offsets rounded.

Offsets are rounded to the preceding (floor) or succeeding (ceil) whole number. The lengths are not changed.

Parameters

- **op** (*str*) – 'ceil' or 'floor'
- **pixel_precision** (*int, optional (default: None)*) – Number of places of rounding precision.

Returns

Return type *Window*

round_shape(*op='floor'*, *pixel_precision=None*)
Return a copy with width and height rounded.

Lengths are rounded to the preceding (floor) or succeeding (ceil) whole number. The offsets are not changed.

Parameters

- **op** (*str*) – 'ceil' or 'floor'
- **pixel_precision** (*int, optional (default: None)*) – Number of places of rounding precision.

Returns

Return type *Window*

row_off

todict()
A mapping of attribute names and values.

Returns

Return type dict

toranges()
Makes an equivalent pair of range tuples

toslices()
Slice objects for use as an ndarray indexer.

Returns **row_slice, col_slice** – A pair of slices in row, column order

Return type slice

width

class rasterio.windows.**WindowMethodsMixin**
Bases: object

Mixin providing methods for window-related calculations. These methods are wrappers for the functionality in *rasterio.windows* module.

A subclass with this mixin MUST provide the following properties: *transform*, *height* and *width*

window(*left*, *bottom*, *right*, *top*, *precision=None*)
Get the window corresponding to the bounding coordinates.

The resulting window is not cropped to the row and column limits of the dataset.

Parameters

- **left** (*float*) – Left (west) bounding coordinate
- **bottom** (*float*) – Bottom (south) bounding coordinate

- **right** (*float*) – Right (east) bounding coordinate
- **top** (*float*) – Top (north) bounding coordinate
- **precision** (*int, optional*) – Number of decimal points of precision when computing inverse transform.

Returns **window**

Return type *Window*

window_bounds(*window*)
Get the bounds of a window

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **bounds** – x_min, y_min, x_max, y_max for the given window

Return type tuple

window_transform(*window*)
Get the affine transform for a dataset window.

Parameters **window** (rasterio.windows.Window) – Dataset window

Returns **transform** – The affine transform matrix for the given window

Return type Affine

rasterio.windows.**bounds**(*window*, *transform*, *height=0*, *width=0*)
Get the spatial bounds of a window.

Parameters

- **window** (Window) – The input window.
- **transform** (*Affine*) – an affine transform matrix.

Returns **left, bottom, right, top** – A tuple of spatial coordinate bounding values.

Return type float

rasterio.windows.**crop**(*window*, *height*, *width*)
Crops a window to given height and width.

Parameters

- **window** (*Window.*) – The input window.
- **height** (*int*) – The number of rows and cols in the cropped window.
- **width** (*int*) – The number of rows and cols in the cropped window.

Returns A new Window object.

Return type *Window*

rasterio.windows.**evaluate**(*window*, *height*, *width*, *boundless=False*)
Evaluates a window tuple that may contain relative index values.

The height and width of the array the window targets is the context for evaluation.

Parameters

- **window** (*Window or tuple of (rows, cols)*) – The input window.
- **height** (*int*) – The number of rows or columns in the array that the window targets.
- **width** (*int*) – The number of rows or columns in the array that the window targets.

Returns A new Window object with absolute index values.

Return type *Window*

rasterio.windows.**from_bounds**(*left*, *bottom*, *right*, *top*, *transform=None*, *height=None*, *width=None*, *precision=None*)

Get the window corresponding to the bounding coordinates.

Parameters

- **left** (*float, required*) – Left (west) bounding coordinates
- **bottom** (*float, required*) – Bottom (south) bounding coordinates
- **right** (*float, required*) – Right (east) bounding coordinates
- **top** (*float, required*) – Top (north) bounding coordinates
- **transform** (*Affine, required*) – Affine transform matrix.
- **height** (*int, required*) – Number of rows of the window.
- **width** (*int, required*) – Number of columns of the window.
- **precision** (*int or float, optional*) – An integer number of decimal points of precision when computing inverse transform, or an absolute float precision.

Returns A new Window.

Return type *Window*

Raises ***WindowError*** – If a window can't be calculated.

rasterio.windows.**get_data_window**(*arr*, *nodata=None*)

Window covering the input array's valid data pixels.

Parameters

- **arr** (*numpy ndarray, <= 3 dimensions*) –
- **nodata** (*number*) – If None, will either return a full window if arr is not a masked array, or will use the mask to determine non-nodata pixels. If provided, it must be a number within the valid range of the dtype of the input array.

Returns

Return type *Window*

rasterio.windows.**intersect**(**windows*)

Test if all given windows intersect.

Parameters **windows** (*sequence*) – One or more Windows.

Returns True if all windows intersect.

Return type bool

rasterio.windows.**intersection**(**windows*)

Innermost extent of window intersections.

Will raise WindowError if windows do not intersect.

Parameters **windows** (*sequence*) – One or more Windows.

Returns

Return type *Window*

rasterio.windows.**iter_args**(*function*)
Decorator to allow function to take either *args or a single iterable which gets expanded to *args.

rasterio.windows.**round_window_to_full_blocks**(*window*, *block_shapes*, *height=0*, *width=0*)
Round window to include full expanse of intersecting tiles.

Parameters

- **window** (Window) – The input window.
- **block_shapes** (*tuple of block shapes*) – The input raster's block shape. All bands must have the same block/stripe structure

Returns

Return type *Window*

rasterio.windows.**shape**(*window*, *height=- 1*, *width=- 1*)
The shape of a window.

height and width arguments are optional if there are no negative values in the window.

Parameters

- **window** (Window) – The input window.
- **height** (*int, optional*) – The number of rows or columns in the array that the window targets.
- **width** (*int, optional*) – The number of rows or columns in the array that the window targets.

Returns The number of rows and columns of the window.

Return type num_rows, num_cols

rasterio.windows.**toranges**(*window*)
Normalize Windows to range tuples

rasterio.windows.**transform**(*window*, *transform*)
Construct an affine transform matrix relative to a window.

Parameters

- **window** (Window) – The input window.
- **transform** (*Affine*) – an affine transform matrix.

Returns The affine transform matrix for the given window

Return type Affine

rasterio.windows.**union**(**windows*)
Union windows and return the outermost extent they cover.

Parameters **windows** (*sequence*) – One or more Windows.

Returns

Return type *Window*

rasterio.windows.**validate_length_value**(*instance*, *attribute*, *value*)

rasterio.windows.**window_index**(*window*, *height=0*, *width=0*)
Construct a pair of slice objects for ndarray indexing

Starting indexes are rounded down, Stopping indexes are rounded up.

Parameters **window** (Window) – The input window.

Returns **row_slice, col_slice** – A pair of slices in row, column order

Return type slice

6.1.3 Module contents

Rasterio

class rasterio.**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.

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.

Returns

Return type None

classmethod default_options()

Default configuration options

Parameters **None** –

Returns

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()

Returns

Return type *Env*

Notes

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

rasterio.**band**(*ds*, *bidx*)
A dataset and one or more of its bands

Parameters

- **ds** (*dataset object*) – An opened rasterio dataset object.
- **bidx** (*int or sequence of ints*) – Band number(s), index starting at 1.

Returns

Return type rasterio.Band

rasterio.**open**(*fp*, *mode='r'*, *driver=None*, *width=None*, *height=None*, *count=None*, *crs=None*, *transform=None*, *dtype=None*, *nodata=None*, *sharing=False*, ***kwargs*)
Open a dataset for reading or writing.

The dataset may be located in a local file, in a resource located by a URL, or contained within a stream of bytes.

In read ('r') or read/write ('r+') mode, no keyword arguments are required: these attributes are supplied by the opened dataset.

In write ('w' or 'w+') mode, the driver, width, height, count, and dtype keywords are strictly required.

Parameters

- **fp** (*str, file object or pathlib.Path object*) – A filename or URL, a file object opened in binary ('rb') mode, or a Path object.
- **mode** (*str, optional*) – 'r' (read, the default), 'r+' (read/write), 'w' (write), or 'w+' (write/read).
- **driver** (*str, optional*) – A short format driver name (e.g. "GTiff" or "JPEG") or a list of such names (see GDAL docs at http://www.gdal.org/formats_list.html). In 'w' or 'w+' modes a single name is required. In 'r' or 'r+' modes the driver can usually be omitted. Registered drivers will be tried sequentially until a match is found. When multiple drivers are available for a format such as JPEG2000, one of them can be selected by using this keyword argument.
- **width** (*int, optional*) – The number of columns of the raster dataset. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.
- **height** (*int, optional*) – The number of rows of the raster dataset. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.
- **count** (*int, optional*) – The count of dataset bands. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.

- **crs** (*str, dict, or CRS; optional*) – The coordinate reference system. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.
- **transform** (*Affine instance, optional*) – Affine transformation mapping the pixel space to geographic space. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.
- **dtype** (*str or numpy dtype*) – The data type for bands. For example: 'uint8' or `rasterio.uint16`. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.
- **nodata** (*int, float, or nan; optional*) – Defines the pixel value to be interpreted as not valid data. Required in 'w' or 'w+' modes, it is ignored in 'r' or 'r+' modes.
- **sharing** (*bool; optional*) – To reduce overhead and prevent programs from running out of file descriptors, rasterio maintains a pool of shared low level dataset handles. When *True* this function will use a shared handle if one is available. Multithreaded programs must avoid sharing and should set *sharing* to *False*.
- **kwargs** (*optional*) – These are passed to format drivers as directives for creating or interpreting datasets. For example: in 'w' or 'w+' modes a *tiled=True* keyword argument will direct the GeoTIFF format driver to create a tiled, rather than striped, TIFF.

Returns

Return type A `DatasetReader` or `DatasetWriter` object.

Examples

To open a GeoTIFF for reading using standard driver discovery and no directives:

```
>>> import rasterio
>>> with rasterio.open('example.tif') as dataset:
...     print(dataset.profile)
```

To open a JPEG2000 using only the JP2OpenJPEG driver:

```
>>> with rasterio.open(
...         'example.jp2', driver='JP2OpenJPEG') as dataset:
...     print(dataset.profile)
```

To create a new 8-band, 16-bit unsigned, tiled, and LZW-compressed GeoTIFF with a global extent and 0.5 degree resolution:

```
>>> from rasterio.transform import from_origin
>>> with rasterio.open(
...         'example.tif', 'w', driver='GTiff', dtype='uint16',
...         width=720, height=360, count=8, crs='EPSG:4326',
...         transform=from_origin(-180.0, 90.0, 0.5, 0.5),
...         nodata=0, tiled=True, compress='lzw') as dataset:
...     dataset.write(...)
```

`rasterio.`**pad**(*array*, *transform*, *pad_width*, *mode=None*, ***kwargs*)

pad array and adjust affine transform matrix.

Parameters

- **array** (*ndarray*) – Numpy ndarray, for best results a 2D array
- **transform** (*Affine transform*) – transform object mapping pixel space to coordinates

- **pad_width** (*int*) – number of pixels to pad array on all four
- **mode** (*str or function*) – define the method for determining padded values

Returns **(array, transform)** – Tuple of new array and affine transform

Return type tuple

Notes

See numpy docs for details on mode and other kwargs: http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.pad.html

CHAPTER

SEVEN

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 12.04? 14.04?)
- 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 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 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.

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 Git Conventions

We use a variant of centralized workflow described in the Git Book. We have no 1.0 release for Rasterio yet and we are tagging and releasing from the master branch. Our post-1.0 workflow is to be decided.

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.

7.7 Code 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 supports Python 2 and Python 3 in the same code base, which is aided by an internal compatibility module named `compat.py`. It functions similarly to the more widely known six but we only use a small portion of the features so it eliminates a dependency.

We strongly prefer code adhering to PEP8.

Tests are mandatory for new features. We use pytest.

We aspire to 100% coverage for Python modules but coverage of the Cython code is a future aspiration (#515).

7.8 Development Environment

Developing Rasterio requires Python 2.7 or any final release after and including 3.4. 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.

7.8.1 Initial Setup

First, clone Rasterio's `git` repo:

```
$ git clone https://github.com/mapbox/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.

7.8.2 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
```

7.8.3 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
```

7.8.4 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]
```

7.8.5 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.

```
$ py.test --cov rasterio --cov-report term-missing
```

A single test file:

```
$ py.test tests/test_band.py
```

A single test:

```
$ py.test tests/test_band.py::test_band
```

7.9 Additional Information

More technical information lives on the wiki.

- https://github.com/mapbox/rasterio/wiki/Development-Guide
- https://github.com/mapbox/rasterio/wiki/Exposing-GDAL-Functionality
- https://github.com/mapbox/rasterio/wiki/Cython-and-GDAL

The long term goal is to consolidate into this document.

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

8.1 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 enviornment, you are likely to see the error message shown above.

8.2 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` environment variable must be set for PROJ to work properly.

Important: Activate your conda environments. The PROJ conda package will set `PROJ_LIB` to the proper value if you activate your conda environment. If you don't activate your conda enviornment, you are likely to see the exception shown above.

8.3 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` 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.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Y

Z