

---

# **buzzard Documentation**

*Release 0.6.5*

**Nicolas Goguey, Hervé Nivon**

**Oct 26, 2020**



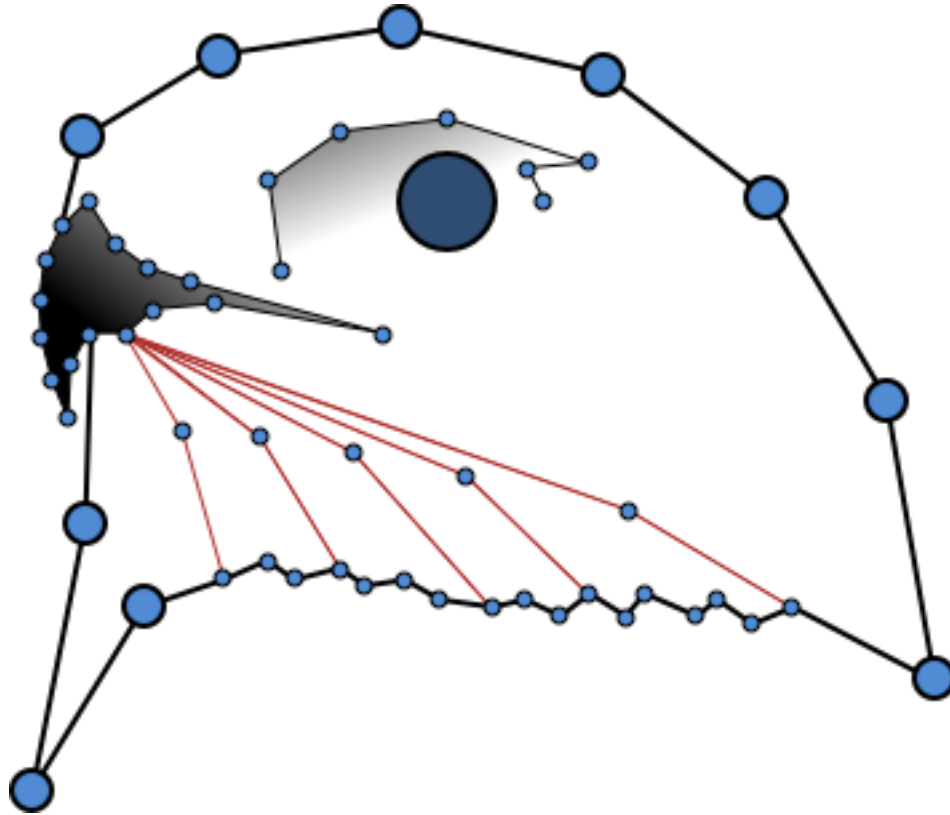
# CONTENTS

<b>1</b>	<b>API</b>	<b>3</b>
1.1	Dataset . . . . .	3
1.2	Sources . . . . .	23
1.3	Footprint . . . . .	59
1.4	Env . . . . .	84
1.5	Misc. . . . .	85
<b>2</b>	<b>Caveats, FAQs and design choices</b>	<b>87</b>
2.1	Caveat List . . . . .	87
2.2	FAQs and design choices . . . . .	88
<b>3</b>	<b>Indices and tables:</b>	<b>91</b>
	<b>Index</b>	<b>93</b>



In a nutshell, buzzard reads and writes geospatial raster and vector data.

Repository is located here: <https://github.com/earthcube-lab/buzzard>





## 1.1 Dataset

### 1.1.1 Dataset

```
class buzzard.Dataset (sr_work=None, sr_fallback=None, sr_forced=None, analyse_transformation=True, allow_none_geometry=False, allow_interpolation=False, max_active=inf, debug_observers=(), **kwargs)
```

**Dataset** is a class that stores references to sources. A source is either a raster, or a vector. A *Dataset* allows:

- quick manipulations by optionally assigning a key to each registered source, (see *Sources Registering* below)
- closing all source at once by closing the Dataset object.

But also inter-sources operations, like:

- spatial reference harmonization (see *On the fly re-projections in buzzard* below),
- workload scheduling on pools when using async rasters (see *Scheduler* below),
- other features in the future (like data visualization).

For actions specific to opened sources, see those classes:

- *GDALFileRaster*
- *GDALMemRaster*
- *NumpyRaster*
- *CachedRasterRecipe*
- *GDALFileVector*
- *GDALMemoryVector*

**Warning:** This class is not equivalent to the *gdal.Dataset* class.

## Parameters

**sr\_work: None or string** In order to set a spatial reference, use a string that can be converted to WKT by GDAL.

(see *On the fly re-projections in buzzard* below)

**sr\_fallback: None or string** In order to set a spatial reference, use a string that can be converted to WKT by GDAL.

(see *On the fly re-projections in buzzard* below)

**sr\_forced: None or string** In order to set a spatial reference, use a string that can be converted to WKT by GDAL.

(see *On the fly re-projections in buzzard* below)

**analyse\_transformation: bool** Whether or not to perform a basic analysis on two *sr* to check their compatibility.

if True: Read the *buzz.env.significant* variable and raise an exception if a spatial reference conversions is too lossy in precision.

if False: Skip all checks.

(see *On the fly re-projections in buzzard* below)

**allow\_none\_geometry: bool** Whether or not a vector geometry should raise an exception when encountering a None geometry

**allow\_interpolation: bool** Whether or not a raster geometry should raise an exception when remapping with interpolation is necessary.

**max\_active: nbr >= 1** Maximum number of pooled sources active at the same time. (see *Sources activation / deactivation* below)

**debug\_observers: sequence of object** Entry points to observe what is happening in the Dataset's sheduler.

## Examples

```
>>> import buzzard as buzz
```

Creating a Dataset.

```
>>> ds = buzz.Dataset()
```

Opening a file and registering it under the 'roofs' key. There are four ways to the access an opened source.

```
>>> r = ds.open_vector('roofs', 'path/to/roofs.shp')
... feature_count = len(ds.roofs)
... feature_count = len(ds['roofs'])
... feature_count = len(ds.get('roofs'))
... feature_count = len(r)
```

Opening a file anonymously. There is only one way to access the source.

```
>>> r = ds.aopen_raster('path/to/dem.tif')
... data_type = r.dtype
```

Opening, reading and closing two raster files with context management.



```
>>> with ds.open_raster('rgb', 'path/to/rgb.tif').close:
...     data_type = ds.rgb.fp
...     arr = ds.rgb.get_data()
```

```
>>> with ds.aopen_raster('path/to/rgb.tif').close as rgb:
...     data_type = rgb.dtype
...     arr = rgb.get_data()
```

### Creating two files

```
>>> ds.create_vector('targets', 'path/to/targets.geojson', 'point', driver=
↳ 'GeoJSON')
... geometry_type = ds.targets.type
```

```
>>> with ds.acreate_raster('/tmp/cache.tif', ds.dem.fp, 'float32', 1).delete as _
↳ cache:
...     file_footprint = cache.fp
...     cache.set_data(dem.get_data())
```

## Sources Types

- **Raster sources**
  - GDAL drivers [http://www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html) (e.g. 'GTiff', 'JPEG', 'PNG', ...)
  - `numpy.ndarray`
  - recipes
- **Vector sources**
  - OGR drivers: [https://www.gdal.org/ogr\\_formats.html](https://www.gdal.org/ogr_formats.html) (e.g. 'ESRI Shapefile', 'GeoJSON', 'DXF', ...)

## Sources Registering

There are always two ways to create a source, with a key or anonymously.

When creating a source using a key, said key (e.g. the string “my\_source\_name”) must be provided by user. Each key identify one source and should thus be unique. There are then three ways to access that source:

- from the object returned by the method that created the source,
- from the Dataset with the attribute syntax: `ds.my_source_name`,
- from the Dataset with the item syntax: `ds[“my_source_name”]`.

All keys should be unique.

When creating a source anonymously you don't have to provide a key, but the only way to access this source is to use the object returned by the method that created the source.

## Sources activation / deactivation

The sources that inherit from *APooledEmissary* (like *GDALFileVector* and *GDALFileRaster*) are flexible about their underlying driver object. Those sources may be temporary deactivated (useful to limit the number of file descriptors active), or activated multiple time at the same time (useful to perform concurrent reads).

Those sources are automatically activated and deactivated given the current needs and constraints. Setting a *max\_active* lower than *np.inf* in the Dataset constructor, will ensure that no more than *max\_active* driver objects are active at the same time, by deactivating the LRU ones.

## On the fly re-projections in buzzard

A Dataset may perform spatial reference conversions on the fly, like a GIS does. Several modes are available, a set of rules define how each mode work. Those conversions concern both read operations and write operations, all are performed by the OSR library.

Those conversions are only performed on vector's data/metadata and raster's Footprints. This implies that classic raster warping is not included (yet) in those conversions, only raster shifting/scaling/rotation work.

The z coordinates of vectors geometries are also converted, on the other hand elevations are not converted in DEM rasters.

If *analyse\_transformation* is set to *True* (default), all coordinates conversions are tested against *buzz.env.significant* on file opening to ensure their feasibility or raise an exception otherwise. This system is naive and very restrictive, use with caution. Although, disabling those tests is not recommended, ignoring floating point precision errors can create unpredictable behaviors at the pixel level deep in your code. Those bugs can be witnessed when zooming to infinity with tools like *qgis* or *matplotlib*.

## On the fly re-projections in buzzard - Terminology

*sr* Spatial reference

*sr\_work* The sr of all interactions with a Dataset (i.e. Footprints, extents, Polygons...), may be None.

*sr\_stored* The sr that can be found in the metadata of a raster/vector storage, may be None.

*sr\_virtual* The sr considered to be written in the metadata of a raster/vector storage, it is often the same as *sr\_stored*. When a raster/vector is read, a conversion is performed from *sr\_virtual* to *sr\_work*. When writing vector data, a conversion is performed from *sr\_work* to *sr\_virtual*.

*sr\_forced* A *sr\_virtual* provided by user to ignore all *sr\_stored*. This is for example useful when the *sr* stored in the input files are corrupted.

*sr\_fallback* A *sr\_virtual* provided by user to be used when *sr\_stored* is missing. This is for example useful when an input file can't store a *sr* (e.g. DFX).

## On the fly re-projections in buzzard - Dataset parameters and modes

mode	sr_work	sr_fallback	sr_forced	How is the <i>sr_virtual</i> of a source determined
1	None	None	None	Use <i>sr_stored</i> , no conversion is performed for the lifetime of this Dataset
2	string	None	None	Use <i>sr_stored</i> , if None raises an exception
3	string	string	None	Use <i>sr_stored</i> , if None it is considered to be <i>sr_fallback</i>
4	string	None	string	Use <i>sr_forced</i>

## On the fly re-projections in buzzard - Use cases

- **If all opened files are known to be written in a same sr in advance, use *mode 1*.** No conversions will be performed, this is the safest way to work.
- **If all opened files are known to be written in the same sr but you wish to work in a different sr, use *mode 4*.** The huge benefit of this mode is that the *driver* specific behaviors concerning spatial references have no impacts on the data you manipulate.
- **On the other hand if you don't have a priori information on files' sr, *mode 2* or *mode 3* should be used.**

**Warning:** Side note: Since the GeoJSON driver cannot store a *sr*, it is impossible to open or create a GeoJSON file in *mode 2*.

## On the fly re-projections in buzzard - Examples

mode 1 - No conversions at all

```
>>> ds = buzz.Dataset()
```

mode 2 - Working with WGS84 coordinates

```
>>> ds = buzz.Dataset (
...     sr_work='WGS84',
... )
```

mode 3 - Working in UTM with DXF files in WGS84 coordinates

```
>>> ds = buzz.Dataset (
...     sr_work='EPSG:32632',
...     sr_fallback='WGS84',
... )
```

mode 4 - Working in UTM with unreliable LCC input files

```
>>> ds = buzz.Dataset (
...     sr_work='EPSG:32632',
...     sr_forced='EPSG:27561',
... )
```

## Scheduler

To handle *async rasters* living in a Dataset, a thread is to manage requests made to those rasters. It will start as soon as you create an *async raster* and stop when the Dataset is closed or collected. If one of your callbacks to be called by the scheduler raises an exception, the scheduler will stop and the exception will be propagated to the main thread as soon as possible.

## Thread-safety

Thread safety is one of the main concern of buzzaard. Everything is thread-safe except:

- The raster write methods
- The vector write methods
- The raster read methods when using the *GDAL::MEM* driver
- The vector read methods when using the *GDAL::Memory* driver

`__del__()`

### property `close`

Close the Dataset with a call or a context management. The *close* attribute returns an object that can be both called and used in a with statement

The Dataset can be closed manually or automatically when garbage collected, it is safer to do it manually.

The internal steps are:

- Stopping the scheduler
- Joining the mp.Pool that have been automatically allocated
- Closing all sources

## Examples

```
>>> ds = buzz.Dataset()
... # code...
... ds.close()
```

```
>>> with buzz.Dataset().close as ds
...     # code...
```

## Caveat

When using a scheduler, some memory leaks may still occur after closing a Dataset. Possible origins:

- <https://bugs.python.org/issue34172> (update your python to >=3.6.7)
- Gdal cache not flushed (not a leak)
- The gdal version
- <https://stackoverflow.com/a/1316799> (not a leak)
- Some unknown leak in the python *threading* or *multiprocessing* standard library
- Some unknown library leaking memory on the C side
- Some unknown library storing data in global variables

You can use a *debug\_observer* with an *on\_object\_allocated* method to track large objects allocated in the scheduler. It will likely not be the source of the problem. If you even find a source of leaks please contact the buzzaard team. <https://github.com/earthcube-lab/buzzaard/issues>

`__getitem__(key)`

Retrieve a source from its key

`__contains__ (item)`

Is key or source registered in Dataset

`items ()`

Generate the pair of (keys\_of\_source, source) for all proxies

`keys ()`

Generate all source keys

`values ()`

Generate all proxies

`__len__ ()`

Retrieve source count registered within this Dataset

`property proj4`

Dataset's work spatial reference in WKT proj4. Returns None if *mode 1*.

`property wkt`

Dataset's work spatial reference in WKT format. Returns None if *mode 1*.

`property active_count`

Count how many driver objects are currently active

`activate_all ()`

Activate all deactivable proxies. May raise an exception if the number of sources is greater than *max\_activated*

`deactivate_all ()`

Deactivate all deactivable proxies. Useful to flush all files to disk

`property pools`

Get the Pool Container.

```
>>> help(PoolsContainer)
```

## 1.1.2 Pool Container

`class buzzard.PoolsContainer`

Manages thread/process pools and aliases for a Dataset

`alias (key, pool_or_none)`

Register the given pool under the given key in this Dataset. The key can then be used to refer to that pool from within the async raster constructors.

### Parameters

**key:** hashable (like a string)

**pool\_or\_none:** `multiprocessing.pool.Pool` or `multiprocessing.pool.ThreadPool` or None

`manage (pool)`

Add the given pool to the list of pools that must be terminated upon Dataset closing.

## Parameters

**pool:** multiprocessing.pool.Pool or multiprocessing.pool.ThreadPool

- `__len__()`  
Number of pools registered in this Dataset
- `__iter__()`  
Generator of pools registered in this Dataset
- `__getitem__(key)`  
Pool or none getter from alias
- `__contains__(obj)`  
Is pool or alias registered in this Dataset

## 1.1.3 Source Constructors

### Rasters Sources Using GDAL

Dataset.**open\_raster**(*key*, *path*, *driver*='GTiff', *options*=(), *mode*='r')  
Open a raster file within this Dataset under *key*. Only metadata are kept in memory.

```
>>> help(GDALFileRaster)
```

## Parameters

**key:** hashable (like a string) File identifier within Dataset

To avoid using a *key*, you may use `aopen_raster()`

**path:** string

**driver:** string gdal driver to use when opening the file [http://www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html)

**options:** sequence of str options for gdal

**mode:** one of {'r', 'w'}

## Returns

**source:** GDALFileRaster

## Example

```
>>> ds.open_raster('ortho', '/path/to/ortho.tif')
>>> file_proj4 = ds.ortho.proj4_stored
```

```
>>> ds.open_raster('dem', '/path/to/dem.tif', mode='w')
>>> no_data_value = ds.dem.no_data
```

## See Also

- `Dataset.aopen_raster()`: To skip the *key* assignment
- `buzzard.open_raster()`: To skip the *key* assignment and the explicit *Dataset* instantiation

`Dataset.create_raster` (*key*, *path*, *fp*, *dtype*, *channel\_count*, *channels\_schema=None*, *driver='GTiff'*, *options=()*, *sr=None*, *ow=False*, *\*\*kwargs*)

Create a raster file and register it under *key* within this *Dataset*. Only metadata are kept in memory.

The raster's values are initialized with *channels\_schema['nodata']* or *0*.

```
>>> help(GDALFileRaster)
>>> help(GDALMemRaster)
```

## Parameters

**key: hashable (like a string)** File identifier within *Dataset*

To avoid using a *key*, you may use `acreate_raster()`

**path: string** Anything that makes sense to GDAL:

- A path to a file
- An empty string when using *driver=MEM*
- A path or an xml string when using *driver=VRT*

**fp: Footprint** Description of the location and size of the raster to create.

**dtype: numpy type (or any alias)**

**channel\_count: integer** number of channels

**channels\_schema: dict or None** Channel(s) metadata. (see *Channels schema fields* below)

**driver: string** gdal driver to use when opening the file [http://www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html)

**options: sequence of str** options for gdal [http://www.gdal.org/frmt\\_gtiff.html](http://www.gdal.org/frmt_gtiff.html)

**sr: string or None** Spatial reference of the new file.

In order not to set a spatial reference, use *None*.

In order to set a spatial reference, use a string that can be converted to WKT by GDAL.

**ow: bool** Overwrite. Whether or not to erase the existing files.

## Returns

**source: GDALFileRaster or GDALMemRaster** The type depends on the *driver* parameter

### Example

```
>>> ds.create_raster('dem_copy', 'dem_copy.tif', ds.dem.fp, ds.dsm.dtype, len(ds.
↳dem) )
>>> array = ds.dem.get_data()
>>> ds.dem_copy.set_data(array)
```

### Channel schema fields

**Fields:** 'nodata': None or number 'interpretation': None or str 'offset': None or number 'scale': None or number 'mask': None or str

**Interpretation values:** undefined, grayindex, paletteindex, redband, greenband, blueband, alphaband, hueband, saturationband, lightnessband, cyanband, magentaband, yellowband, blackband

**Mask values:** all\_valid, per\_dataset, alpha, nodata

Additionally:

- A field missing or None is kept to default value.
- A field can be passed as
  - a value: All bands are set to this value
  - a sequence of values of length *channel\_count*: All bands will be set to their respective state

### Caveat

When using the GTiff driver, specifying a *mask* or *interpretation* field may lead to unexpected results.

### See Also

- `Dataset.create_raster()`: To skip the *key* assignment
- `buzzaard.create_raster()`: To skip the *key* assignment and the explicit *Dataset* instantiation

`Dataset.aopen_raster` (*path*, *driver*='GTiff', *options*=(), *mode*='r')

Open a raster file anonymously within this Dataset. Only metadata are kept in memory.

See `open_raster()`

### Example

```
>>> ortho = ds.aopen_raster('/path/to/ortho.tif')
>>> file_wkt = ortho.wkt_stored
```



## See Also

- `Dataset.open_raster()`: To assign a *key* to this source within the *Dataset*
- `buzzard.open_raster()`: To skip the explicit *Dataset* instantiation

`Dataset.acreate_raster` (*path, fp, dtype, channel\_count, channels\_schema=None, driver='GTiff, options=(), sr=None, ow=False, \*\*kwargs*)  
Create a raster file anonymously within this Dataset. Only metadata are kept in memory.

See `create_raster()`

## Example

```
>>> mask = ds.acreate_raster('mask.tif', ds.dem.fp, bool, 1, options=['SPARSE_
↳OK=YES'])
>>> open_options = mask.open_options
```

```
>>> channels_schema = {
...     'nodata': -32767,
...     'interpretation': ['blackband', 'cyanband'],
... }
>>> out = ds.acreate_raster('output.tif', ds.dem.fp, 'float32', 2, channels_
↳schema)
>>> band_interpretation = out.channels_schema['interpretation']
```

## See Also

- `Dataset.create_raster()`: To assign a *key* to this source within the *Dataset*
- `buzzard.create_raster()`: To skip the explicit *Dataset* instantiation

## Rasters Sources Using NumPy

`Dataset.wrap_numpy_raster` (*key, fp, array, channels\_schema=None, sr=None, mode='w', \*\*kwargs*)  
Register a numpy array as a raster under *key* within this Dataset.

```
>>> help(NumpyRaster)
```

## Parameters

**key:** hashable (like a string) File identifier within Dataset

To avoid using a *key*, you may use `awrap_numpy_raster()`

**fp:** Footprint of shape (Y, X) Description of the location and size of the raster to create.

**array:** ndarray of shape (Y, X) or (Y, X, C)

**channels\_schema:** dict or None Channel(s) metadata. (see *Channels schema fields* below)

**sr:** string or None Spatial reference of the new file

In order not to set a spatial reference, use *None*.

In order to set a spatial reference, use a string that can be converted to WKT by GDAL.

## Returns

source: `NumpyRaster`

## Channel schema fields

**Fields:** 'nodata': None or number 'interpretation': None or str 'offset': None or number 'scale': None or number 'mask': None or str

**Interpretation values:** undefined, grayindex, paletteindex, redband, greenband, blueband, alphaband, hueband, saturationband, lightnessband, cyanband, magentaband, yellowband, blackband

**Mask values:** all\_valid, per\_dataset, alpha, nodata

Additionally:

- A field missing or None is kept to default value.
- A field can be passed as
  - a value: All bands are set to this value
  - a sequence of values of length `channel_count`: All bands will be set to their respective state

## See Also

- `Dataset.wrap_numpy_raster()`: To skip the *key* assignment
- `buzzard.wrap_numpy_raster()`: To skip the *key* assignment and the explicit `Dataset` instantiation

`Dataset.wrap_numpy_raster` (*fp*, *array*, *channels\_schema=None*, *sr=None*, *mode='w'*, *\*\*kwargs*)  
 Register a numpy array as a raster anonymously within this Dataset.

## See Also

- `Dataset.wrap_numpy_raster()`: To assign a *key* to this source within the `Dataset`
- `buzzard.wrap_numpy_raster()`: To skip the *key* assignment and the explicit `Dataset` instantiation

## Rasters Sources Using Recipes

`Dataset.create_raster_recipe` (*key*, *fp*, *dtype*, *channel\_count*, *channels\_schema=None*, *sr=None*, *compute\_array=None*, *merge\_arrays=<function concat\_arrays>*, *queue\_data\_per\_primitive=mappingproxy({})*, *convert\_footprint\_per\_primitive=None*, *computation\_pool='cpu'*, *merge\_pool='cpu'*, *resample\_pool='cpu'*, *computation\_tiles=None*, *max\_computation\_size=None*, *max\_resampling\_size=None*, *automatic\_remapping=True*, *debug\_observers=()*)

**Warning:** This method is not yet implemented. It exists for documentation purposes.

Create a *raster recipe* and register it under *key* within this Dataset.

A *raster recipe* implements the same interfaces as all other rasters, but internally it computes data on the fly by calling a callback. The main goal of the *raster recipes* is to provide a boilerplate-free interface that automatize those cumbersome tasks:

- tiling,
- parallelism
- caching
- file reads
- resampling
- lazy evaluation
- backpressure prevention and
- optimised task scheduling.

If you are familiar with `create_cached_raster_recipe` two parameters are new here: `automatic_remapping` and `max_computation_size`.

## Parameters

**key:** see `Dataset.create_raster()`

**fp:** see `Dataset.create_raster()`

**dtype:** see `Dataset.create_raster()`

**channel\_count:** see `Dataset.create_raster()`

**channels\_schema:** see `Dataset.create_raster()`

**sr:** see `Dataset.create_raster()`

**compute\_array:** callable see *Computation Function* below

**merge\_arrays:** callable see *Merge Function* below

**queue\_data\_per\_primitive:** dict of hashable (like a string) to a *queue\_data* method pointer see *Primitives* below

**convert\_footprint\_per\_primitive:** None or dict of hashable (like a string) to a callable see *Primitives* below

**computation\_pool:** see *Pools* below

**merge\_pool:** see *Pools* below

**resample\_pool:** see *Pools* below

**computation\_tiles:** None or (int, int) or numpy.ndarray of Footprint see *Computation Tiling* below

**max\_computation\_size:** None or int or (int, int) see *Computation Tiling* below

**max\_resampling\_size:** None or int or (int, int) Optionally define a maximum resampling size. If a larger resampling has to be performed, it will be performed tile by tile in parallel.

**automatic\_remapping:** bool see *Automatic Remapping* below

**debug\_observers:** sequence of object Entry points that observe what is happening with this raster in the Dataset's scheduler.

## Returns

source: `NocacheRasterRecipe`

## Computation Function

The function that will map a Footprint to a `numpy.ndarray`. If `queue_data_per_primitive` is not empty, it will map a Footprint and primitive arrays to a `numpy.ndarray`.

It will be called in parallel according to the `computation_pool` parameter provided at construction.

The function will be called with the following positional parameters:

- **fp: Footprint of shape (Y, X)** The location at which the pixels should be computed
- **primitive\_fps: dict of hashable to Footprint** For each primitive defined through the `queue_data_per_primitive` parameter, the input Footprint.
- **primitive\_arrays: dict of hashable to numpy.ndarray** For each primitive defined through the `queue_data_per_primitive` parameter, the input `numpy.ndarray` that was automatically computed.
- **raster: CachedRasterRecipe or None** The Raster object of the ongoing computation.

It should return either:

- **a single ndarray of shape (Y, X) if only one channel was computed**
- **a single ndarray of shape (Y, X, C) if one or more channels were computed**

If `computation_pool` points to a process pool, the `compute_array` function must be picklable and the `raster` parameter will be `None`.

## Computation Tiling

You may sometimes want to have control on the Footprints that are requested to the `compute_array` function, for example:

- If pixels computed by `compute_array` are long to compute, you want to tile to increase parallelism.
- If the `compute_array` function scales badly in term of memory or time, you want to tile to reduce complexity.
- If `compute_array` can work only on certain Footprints, you want a hard constraint on the set of Footprint that can be queried from `compute_array`. (This may happen with *convolutional neural networks*)

To do so use the `computation_tiles` or `max_computation_size` parameter (not both).

If `max_computation_size` is provided, a Footprint to be computed will be tiled given this parameter.

If `computation_tiles` is a `numpy.ndarray` of Footprint, it should be a tiling of the `fp` parameter. Only the Footprints contained in this tiling will be asked to the `computation_tiles`. If `computation_tiles` is (int, int), a tiling will be constructed using `Footprint.tile` using those two ints.

## Merge Function

The function that will map several pairs of Footprint/numpy.ndarray to a single numpy.ndarray. If the *computation\_tiles* is None, it will never be called.

It will be called in parallel according to the *merge\_pool* parameter provided at construction.

The function will be called with the following positional parameters:

- **fp: Footprint of shape (Y, X)** The location at which the pixels should be computed.
- **array\_per\_fp: dict of Footprint to numpy.ndarray** The pairs of Footprint/numpy.ndarray of each arrays that were computed by *compute\_array* and that overlap with *fp*.
- **raster: CachedRasterRecipe or None** The Raster object of the ongoing computation.

It should return either:

- **a single ndarray of shape (Y, X) if only one channel was computed**
- **a single ndarray of shape (Y, X, C) if one or more channels were computed**

If *merge\_pool* points to a process pool, the *merge\_array* function must be picklable and the *raster* parameter will be None.

## Automatic Remapping

When creating a recipe you give a *Footprint* through the *fp* parameter. When calling your *compute\_array* function the scheduler will only ask for slices of *fp*. This means that the scheduler takes care of those boilerplate steps:

- If you request a *Footprint* on a different grid in a *get\_data()* call, the scheduler **takes care of resampling** the outputs of your *compute\*array* function.
- If you request a *Footprint* partially or fully outside of the raster's extent, the scheduler will call your *compute\_array* function to get the interior pixels and then **pad the output with nodata**.

This system is flexible and can be deactivated by passing *automatic\_remapping=False* to the constructor of a *NocacheRasterRecipe*, in this case the scheduler will call your *compute\_array* function for any kind of *Footprint*; thus your function must be able to comply with any request.

## Primitives

The *queue\_data\_per\_primitive* and *convert\_footprint\_per\_primitive* parameters can be used to create dependencies between *dependee async rasters* and the *raster recipe* being created. The dependee/dependent relation is called primitive/derived throughout buzzard. A derived recipe can itself be the primitive of another raster. Pipelines of any depth and width can be instantiated that way.

In *queue\_data\_per\_primitive* you declare a *dependee* by giving it a key of your choice and the pointer to the *queue\_data* method of *dependee* raster. You can parameterize the connection by *currying* the *channels*, *dst\_nodata*, *interpolation* and *max\_queue\_size* parameters using *functools.partial*.

The *convert\_footprint\_per\_primitive* dict should contain the same keys as *queue\_data\_per\_primitive*. A value in the dict should be a function that maps a Footprint to another Footprint. It can be used for example to request larger rectangles of primitives data to compute a derived array.

e.g. If the primitive raster is an *rgb* image, and the derived raster only needs the green channel but with a context of 10 additional pixels on all 4 sides:

```
>>> derived = ds.create_raster_recipe(
...     # <other parameters>
...     queue_data_per_primitive={'green': functools.partial(primitive.queue_data,
→ channels=1)},
...     convert_footprint_per_primitive={'green': lambda fp: fp.dilate(10)},
... )
```

## Pools

The `*_pool` parameters can be used to select where certain computations occur. Those parameters can be of the following types:

- A *multiprocessing.pool.ThreadPool*, should be the default choice.
- A *multiprocessing.pool.Pool*, a process pool. Useful for computations that requires the GIL or that leaks memory.
- *None*, to request the scheduler thread to perform the tasks itself. Should be used when the computation is very light.
- A *hashable* (like a *string*), that will map to a pool registered in the *Dataset*. If that key is missing from the *Dataset*, a *ThreadPool* with `multiprocessing.cpu_count()` workers will be automatically instantiated. When the *Dataset* is closed, the pools instantiated that way will be joined.

## See Also

- `Dataset.create_raster_recipe()`: To skip the *key* assignment
- `Dataset.create_raster_recipe()`: For results *caching*
- `Dataset.create_cached_raster_recipe()`: To skip the *key* assignment

```
Dataset.create_cached_raster_recipe(key, fp, dtype, channel_count, channels_schema=None, sr=None, compute_array=None, merge_arrays=<function concat_arrays>, cache_dir=None, ow=False, queue_data_per_primitive=mappingproxy({}), convert_footprint_per_primitive=None, computation_pool='cpu', merge_pool='cpu', io_pool='io', resample_pool='cpu', cache_tiles=(512, 512), computation_tiles=None, max_resampling_size=None, debug_observers=())
```

Create a *cached raster recipe* and register it under *key* within this *Dataset*.

Compared to a *NocacheRasterRecipe*, in a *CachedRasterRecipe* the pixels are never computed twice. Cache files are used to store and reuse pixels from computations. The cache can even be reused between python sessions.

If you are familiar with `create_raster_recipe` four parameters are new here: `io_pool`, `cache_tiles`, `cache_dir` and `ow`. They are all related to file system operations.

See `create_raster_recipe` method, since it shares most of the features:

```
>>> help(CachedRasterRecipe)
```

## Parameters

**key:** see `Dataset.create_raster()` method

**fp:** see `Dataset.create_raster()` method

**dtype:** see `Dataset.create_raster()` method

**channel\_count:** see `Dataset.create_raster()` method

**channels\_schema:** see `Dataset.create_raster()` method

**sr:** see `Dataset.create_raster()` method

**compute\_array:** see `Dataset.create_raster_recipe()` method

**merge\_arrays:** see `Dataset.create_raster_recipe()` method

**cache\_dir:** **str or pathlib.Path** Path to the directory that holds the cache files associated with this raster. If cache files are present, they will be reused (or erased if corrupted). If a cache file is needed and missing, it will be computed.

**ow:** **bool** Overwrite. Whether or not to erase the old cache files contained in `cache_dir`.

**Warning:** not only the files needed (hence computed) but all buzzard cache files in `cache_dir` will be deleted.

**queue\_data\_per\_primitive:** see `Dataset.create_raster_recipe()` method

**convert\_footprint\_per\_primitive:** see `Dataset.create_raster_recipe()` method

**computation\_pool:** see `Dataset.create_raster_recipe()` method

**merge\_pool:** see `Dataset.create_raster_recipe()` method

**io\_pool:** see `Dataset.create_raster_recipe()` method

**resample\_pool:** see `Dataset.create_raster_recipe()` method

**cache\_tiles:** **(int, int) or numpy.ndarray of Footprint** A tiling of the `fp` parameter. Each tile will correspond to one cache file. if (int, int): Construct the tiling by calling `Footprint.tile` with this parameter

**computation\_tiles:** if None: Use the same tiling as `cache_tiles` else: see `create_raster_recipe` method

**max\_resampling\_size:** **None or int or (int, int)** see `Dataset.create_raster_recipe()` method

**debug\_observers:** **sequence of object** see `Dataset.create_raster_recipe()` method

## Returns

**source:** `CachedRasterRecipe`

### See Also

- `Dataset.create_raster_recipe()`: To skip the *catching*
- `Dataset.create_cached_raster_recipe()`: To skip the *key* assignment

```
Dataset.create_cached_raster_recipe(fp, dtype, channel_count, channels_schema=None, sr=None, compute_array=None, merge_arrays=<function concat_arrays>, cache_dir=None, ow=False, queue_data_per_primitive=mappingproxy({}), convert_footprint_per_primitive=None, computation_pool='cpu', merge_pool='cpu', io_pool='io', resample_pool='cpu', cache_tiles=(512, 512), computation_tiles=None, max_resampling_size=None, debug_observers=())
```

Create a cached raster recipe anonymously within this Dataset.

See `Dataset.create_cached_raster_recipe`

### See Also

- `Dataset.create_raster_recipe()`: To skip the *catching*
- `Dataset.create_cached_raster_recipe()`: To assign a *key* to this source within the *Dataset*

## Vectors Sources Using GDAL (OGR)

```
Dataset.open_vector(key, path, layer=None, driver='ESRI Shapefile', options=(), mode='r')
```

Open a vector file within this Dataset under *key*. Only metadata are kept in memory.

```
>>> help(GDALFileVector)
```

### Parameters

**key:** hashable (like a string) File identifier within Dataset

To avoid using a *key*, you may use `aopen_vector()`

**path:** string

**layer:** None or int or string

**driver:** string ogr driver to use when opening the file [http://www.gdal.org/ogr\\_formats.html](http://www.gdal.org/ogr_formats.html)

**options:** sequence of str options for ogr

**mode:** one of {'r', 'w'}



## Returns

source: `GDALFileVector`

## Example

```
>>> ds.open_vector('trees', '/path/to.shp')
>>> feature_count = len(ds.trees)
```

```
>>> ds.open_vector('roofs', '/path/to.json', driver='GeoJSON', mode='w')
>>> fields_list = ds.roofs.fields
```

## See Also

- `Dataset.aopen_vector()`: To skip the key assignment
- `buzzard.open_vector()`: To skip the key assignment and the explicit `Dataset` instantiation

`Dataset.aopen_vector` (*path*, *layer=None*, *driver='ESRI Shapefile'*, *options=()*, *mode='r'*)

Open a vector file anonymously within this `Dataset`. Only metadata are kept in memory.

See `open_vector()`

## Example

```
>>> trees = ds.aopen_vector('/path/to.shp')
>>> features_bounds = trees.bounds
```

## See Also

- `Dataset.open_vector()`: To assign a key to this source within the `Dataset`
- `buzzard.open_vector()`: To skip the key assignment and the explicit `Dataset` instantiation

`Dataset.create_vector` (*key*, *path*, *type*, *fields=()*, *layer=None*, *driver='ESRI Shapefile'*, *options=()*, *sr=None*, *ow=False*)

Create an empty vector file and register it under *key* within this `Dataset`. Only metadata are kept in memory.

```
>>> help(GDALFileVector)
>>> help(GDALMemoryVector)
```

## Parameters

**key:** **hashable (like a string)** File identifier within `Dataset`

To avoid using a *key*, you may use `acreate_vector()`

**path:** **string** Anything that makes sense to GDAL:

- A path to a file
- An empty string when using `driver=Memory`

**type:** **string** name of a wkb geometry type, without the *wkb* prefix.

list: [http://www.gdal.org/ogr\\_\\_core\\_8h.html#a800236a0d460ef66e687b7b65610f12a](http://www.gdal.org/ogr__core_8h.html#a800236a0d460ef66e687b7b65610f12a)

**fields:** **sequence of dict** Attributes of fields, one dict per field. (see *Field Attributes* below)

**layer:** **None or string**

**driver:** **string** ogr driver to use when opening the file [http://www.gdal.org/ogr\\_formats.html](http://www.gdal.org/ogr_formats.html)

**options:** **sequence of str** options for ogr

**sr:** **string or None** Spatial reference of the new file

In order not to set a spatial reference, use *None*.

In order to set a spatial reference, use a string that can be converted to WKT by GDAL.

**ow:** **bool** Overwrite. Whether or not to erase the existing files.

## Returns

**source:** **GDALFileVector or GDALMemoryVector** The type depends on the *driver* parameter

## Example

```
>>> ds.create_vector('lines', '/path/to.shp', 'linestring')
>>> geometry_type = ds.lines.type
>>> ds.lines.insert_data([[0, 0], [1, 1], [1, 2]])
```

```
>>> fields = [
    {'name': 'name', 'type': str},
    {'name': 'count', 'type': 'int32'},
    {'name': 'area', 'type': np.float64, 'width': 5, 'precision': 18},
    {'name': 'when', 'type': np.datetime64},
]
>>> ds.create_vector('zones', '/path/to.shp', 'polygon', fields)
>>> field0_type = ds.zones.fields[0]['type']
>>> ds.zones.insert_data(shapely.geometry.box(10, 10, 15, 15))
```

## Field Attributes

Attributes:

- “name”: string
- “type”: string (see *Field Types* below)
- “precision”: int
- “width”: int
- “nullable”: bool
- “default”: same as *type*

An attribute missing or *None* is kept to default value.

## Field Types

Type	Type names
Binary	“binary”, bytes, np.bytes_, aliases of np.bytes_
Date	“date”
DateTime	“datetime”, datetime.datetime, np.datetime64, aliases of np.datetime64
Time	“time”
Integer	“integer” np.int32, aliases of np.int32
Integer64	“integer64”, int, np.int64, aliases of np.int64
Real	“real”, float, np.float64, aliases of np.float64
String	“string”, str, np.str_, aliases of np.str_
Integer64List	“integer64list”, “int list”
IntegerList	“integerlist”
RealList	“reallist”, “float list”

### See Also

- `Dataset.create_vector()`: To skip the *key* assignment
- `buzzard.create_vector()`: To skip the *key* assignment and the explicit *Dataset* instantiation

`Dataset.create_vector(path, type, fields=(), layer=None, driver='ESRI Shapefile', options=(), sr=None, ow=False)`

Create a vector file anonymously within this Dataset. Only metadata are kept in memory.

See `create_vector()`

### Example

```
>>> lines = ds.create_vector('/path/to.shp', 'linestring')
>>> file_proj4 = lines.proj4_stored
```

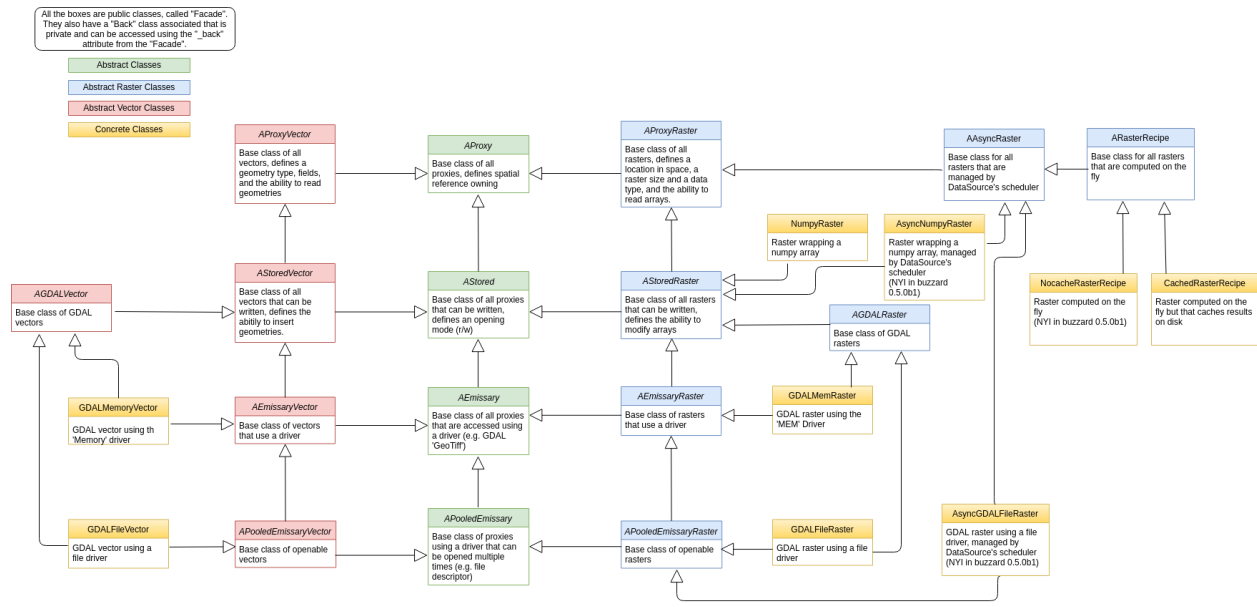
### See Also

- `Dataset.create_vector()`: To assign a *key* to this source within the *Dataset*
- `buzzard.create_vector()`: To skip the *key* assignment and the explicit *Dataset* instantiation

## 1.2 Sources

All sources in buzzard can only be constructed from the Dataset methods, see *Source Constructors*

All sources in buzzard inherit from a series of abstract classes:



## 1.2.1 GDALFileRaster

**class** `buzzard.ASource` (*<implementation detail>*)

Base abstract class defining the common behavior of all sources opened in the Dataset.

### Features Defined

- Has a *stored* spatial reference
- Has a *virtual* spatial reference that is influenced by the Dataset's opening mode
- Can be closed

#### property `wkt_stored`

The spatial reference that can be found in the metadata of a source, in wkt format.

string or None

#### property `proj4_stored`

The spatial reference that can be found in the metadata of a source, in proj4 format.

string or None

#### property `wkt_virtual`

The spatial reference considered to be written in the metadata of a source, in wkt format.

string or None

#### property `proj4_virtual`

The spatial reference considered to be written in the metadata of a source, in proj4 format.

string or None

#### method `get_keys` ()

Get the list of keys under which this source is registered to in the Dataset

**property close**

Close a source with a call or a context management. The *close* attribute returns an object that can be both called and used in a with statement

**Examples**

```
>>> ds.dem.close()
>>> with ds.dem.close:
    # code...
>>> with ds.create_raster('result.tif', fp, float, 1).close as result:
    # code...
>>> with ds.create_vector('results.shp', 'linestring').close as roofs:
    # code...
```

**\_\_del\_\_()**

**class** buzzard.ASourceRaster (<implementation detail>)

Base abstract class defining the common behavior of all rasters.

**Features Defined**

- Has a *stored* Footprint that defines the location of the raster
- Has a Footprint that is influenced by the Dataset's opening mode
- Has a length that defines how many channels are available
- Has a *channels\_schema* that defines per channel attributes (e.g. nodata)
- Has a *dtype* (like np.float32)
- Has a *get\_data* method that allows to read pixels in their current state to numpy arrays

**property fp\_stored****property fp****property channels\_schema****property dtype****property nodata**

Accessor for first channel's nodata value

**get\_nodata** (*channel=0*)

Accessor for nodata value

**\_\_len\_\_()**

Return the number of channels

**get\_data** (*fp=None, channels=None, dst\_nodata=None, interpolation='cv\_area', \*\*kwargs*)

Read a rectangle of data on several channels from the source raster.

If *fp* is not fully within the source raster, the external pixels are set to nodata. If nodata is missing, 0 is used. If *fp* is not on the same grid as the source raster, remapping is performed using *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to False in *Dataset* (default)). When remapping, the nodata values are not interpolated, they are correctly spread to the output.

If *dst\_nodata* is provided, nodata pixels are set to *dst\_nodata*.

**Warning:** The alpha channels are currently resampled like any other channels, this behavior may change in the future. To normalize an *rgba* array after a resampling operation, use this piece of code:

```
>>> arr = np.where(arr[..., -1] == 255, arr, 0)
```

**Warning:** Bands in GDAL are indexed from 1. Channels in buzzard are indexed from 0.

## Parameters

**fp: Footprint of shape (Y, X) or None** If None: return the full source raster

If Footprint: return this window from the raster

**channels: None or int or slice or sequence of int (see *Channels Parameter* below)** The channels to be read

**dst\_nodata: nbr or None** nodata value in output array If None and raster.nodata is not None: raster.nodata is used If None and raster.nodata is None: 0 is used

**interpolation: one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None**  
OpenCV method used if interpolation is necessary

## Returns

**array: numpy.ndarray of shape (Y, X) or (Y, X, C)**

- If the *channels* parameter is *-1*, the returned array is of shape (Y, X) when *C=1*, (Y, X, C) otherwise.
- If the *channels* parameter is an integer  $\geq 0$ , the returned array is of shape (Y, X).
- If the *channels* parameter is a sequence or a slice, the returned array is always of shape (Y, X, C), no matter the size of *C*.

(see *Channels Parameter* below)

## Channels Parameter

type	value	meaning	output shape
NoneType	None (default)	All channels	(Y, X) or (Y, X, C)
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels	(Y, X, C)
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>	(Y, X)
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels	(Y, X, C)

**class** buzzard.AStored (<implementation detail>)

Base abstract class defining the common behavior of all sources that are stored somewhere (like RAM or disk).

## Features Defined

- Has an opening mode

### property mode

Open mode, one of {'r', 'w'}

**class** `buzzard.AStoredRaster` (<implementation detail>)

Base abstract class defining the common behavior of all rasters that are stored somewhere (like RAM or disk).

## Features Defined

- Has a `set_data` method that allows to write pixels to storage

**set\_data** (*array*, *fp=None*, *channels=None*, *interpolation='cv\_area'*, *mask=None*, *\*\*kwargs*)

Write a rectangle of data to the destination raster. Each channel in *array* is written to one channel in *raster* in the same order as described by the *channels* parameter. An optional *mask* may be provided to only write certain pixels of *array*.

If *fp* is not fully within the destination raster, only the overlapping pixels are written. If *fp* is not on the same grid as the destination raster, remapping is automatically performed using the *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to *False* in *Dataset* (default)). When interpolating:

- The nodata values are not interpolated, they are correctly spread to the output.
- At most one pixel may be lost at edges due to interpolation. Provide more context in *array* to compensate this loss.
- The mask parameter is also interpolated.

The alpha bands are currently resampled like any other band, this behavior may change in the future.

This method is not thread-safe.

## Parameters

**array:** `numpy.ndarray` of shape (Y, X) or (Y, X, C) The values to be written

**fp:** Footprint of shape (Y, X) or None If None: write the full source raster If Footprint: write this window to the raster

**channels:** None or int or slice or sequence of int (see *Channels Parameter* below) The channels to be written.

**interpolation:** one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None  
OpenCV method used if interpolation is necessary

**mask:** `numpy array` of shape (Y, X) and dtype *bool* OR inputs accepted by *Footprint.burn\_polygons*

### Channels Parameter

type	value	meaning
NoneType	None (default)	All channels
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels

### Caveat

When using a Raster backed by a driver (like a GDAL driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Raster.

**fill** (*value*, *channels=None*, *\*\*kwargs*)  
 Fill raster with value.

This method is not thread-safe.

### Parameters

**value:** nbr

**channels:** int or sequence of int (see *Channels Parameter* below) The channels to be written

### Channels Parameter

type	value	meaning
NoneType	None (default)	All channels
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels

### Caveat

When using a Raster backed by a driver (like a GDAL driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Raster.

**class** `buzzard.AEmissary` (<implementation detail>)

Base abstract class defining the common behavior of all sources that are backed by a driver.



## Features Defined

- Has a *driver* (like “GTiff” for GDAL’s geotiff driver)
- Has *open\_options*
- Has a *path* (if the driver supports it)
- Can be deleted (if the driver supports it)

### property driver

Get the driver name, such as ‘GTiff’ or ‘GeoJSON’

### property open\_options

Get the list of options used for opening

### property path

Get the file system path of this source, may be the empty string if not applicable

### property delete

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = ‘r’ The *delete* attribute returns an object that can be both called and used in a with statement

## Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

### property remove

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = ‘r’ The *delete* attribute returns an object that can be both called and used in a with statement

## Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

**class** buzzard.**AEmissaryRaster** (<implementation detail>)

Base abstract class defining the common behavior of all rasters that are backed by a driver.

### Features Defined

None

**class** `buzzard.APooledEmissary` (<implementation detail>)

Base abstract class defining the common behavior of all sources that can deactivate and reactivate their underlying driver at will.

This is useful to balance the number of active file descriptors. This is useful to perform concurrent reads if the driver does not support it.

### Features Defined

- An *activate* method to manually open the driver (Mostly useless feature since opening is automatic if necessary)
- A *deactivate* method to close the driver (Useful to flush data to disk)
- An *active\_count* property
- An *active* property

**activate** ()

Make sure that at least one driver object is active for this Raster/Vector

**deactivate** ()

Collect all active driver object for this Raster/Vector. If a driver object is currently being used, will raise an exception.

**property active\_count**

Count how many driver objects are currently active for this Raster/Vector

**property active**

Is there any driver object currently active for this Raster/Vector

**class** `buzzard.APooledEmissaryRaster` (<implementation detail>)

Base abstract class defining the common behavior of all rasters that can deactivate and reactivate their underlying driver at will.

### Features Defined

None

**class** `buzzard.GDALFileRaster` (<implementation detail>)

Concrete class defining the behavior of a GDAL raster using a file.

```
>>> help(Dataset.open_raster)
>>> help(Dataset.create_raster)
```

## Features Defined

None

## 1.2.2 GDALMemRaster

**class** `buzzard.ASource` (<implementation detail>)

Base abstract class defining the common behavior of all sources opened in the Dataset.

### Features Defined

- Has a *stored* spatial reference
- Has a *virtual* spatial reference that is influenced by the Dataset's opening mode
- Can be closed

#### **property** `wkt_stored`

The spatial reference that can be found in the metadata of a source, in wkt format.

string or None

#### **property** `proj4_stored`

The spatial reference that can be found in the metadata of a source, in proj4 format.

string or None

#### **property** `wkt_virtual`

The spatial reference considered to be written in the metadata of a source, in wkt format.

string or None

#### **property** `proj4_virtual`

The spatial reference considered to be written in the metadata of a source, in proj4 format.

string or None

#### **get\_keys** ()

Get the list of keys under which this source is registered to in the Dataset

#### **property** `close`

Close a source with a call or a context management. The `close` attribute returns an object that can be both called and used in a with statement

### Examples

```
>>> ds.dem.close()
>>> with ds.dem.close:
    # code...
>>> with ds.create_raster('result.tif', fp, float, 1).close as result:
    # code...
>>> with ds.create_vector('results.shp', 'linestring').close as roofs:
    # code...
```

#### **\_\_del\_\_** ()

**class** `buzzard.ASourceRaster` (<implementation detail>)

Base abstract class defining the common behavior of all rasters.

## Features Defined

- Has a *stored* Footprint that defines the location of the raster
- Has a Footprint that is influenced by the Dataset's opening mode
- Has a length that defines how many channels are available
- Has a *channels\_schema* that defines per channel attributes (e.g. nodata)
- Has a *dtype* (like `np.float32`)
- Has a *get\_data* method that allows to read pixels in their current state to numpy arrays

**property** `fp_stored`

**property** `fp`

**property** `channels_schema`

**property** `dtype`

**property** `nodata`

Accessor for first channel's nodata value

**get\_nodata** (*channel=0*)

Accessor for nodata value

**\_\_len\_\_** ()

Return the number of channels

**get\_data** (*fp=None, channels=None, dst\_nodata=None, interpolation='cv\_area', \*\*kwargs*)

Read a rectangle of data on several channels from the source raster.

If *fp* is not fully within the source raster, the external pixels are set to nodata. If nodata is missing, 0 is used. If *fp* is not on the same grid as the source raster, remapping is performed using *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to False in *Dataset* (default)). When remapping, the nodata values are not interpolated, they are correctly spread to the output.

If *dst\_nodata* is provided, nodata pixels are set to *dst\_nodata*.

**Warning:** The alpha channels are currently resampled like any other channels, this behavior may change in the future. To normalize an *rgba* array after a resampling operation, use this piece of code:

```
>>> arr = np.where(arr[..., -1] == 255, arr, 0)
```

**Warning:** Bands in GDAL are indexed from 1. Channels in buzard are indexed from 0.

## Parameters

**fp: Footprint of shape (Y, X) or None** If None: return the full source raster

If Footprint: return this window from the raster

**channels: None or int or slice or sequence of int (see *Channels Parameter* below)** The channels to be read

**dst\_nodata: nbr or None** nodata value in output array If None and raster.nodata is not None: raster.nodata is used If None and raster.nodata is None: 0 is used

**interpolation: one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None**  
OpenCV method used if interpolation is necessary

## Returns

**array: numpy.ndarray of shape (Y, X) or (Y, X, C)**

- If the *channels* parameter is *-1*, the returned array is of shape (Y, X) when *C=1*, (Y, X, C) otherwise.
- If the *channels* parameter is an integer  $\geq 0$ , the returned array is of shape (Y, X).
- If the *channels* parameter is a sequence or a slice, the returned array is always of shape (Y, X, C), no matter the size of *C*.

(see *Channels Parameter* below)

## Channels Parameter

type	value	meaning	output shape
NoneType	None (default)	All channels	(Y, X) or (Y, X, C)
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels	(Y, X, C)
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>	(Y, X)
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels	(Y, X, C)

**class** `buzzaard.AStored` (<implementation detail>)

Base abstract class defining the common behavior of all sources that are stored somewhere (like RAM or disk).

### Features Defined

- Has an opening mode

#### property mode

Open mode, one of {'r', 'w'}

**class** `buzzaard.AStoredRaster` (<implementation detail>)

Base abstract class defining the common behavior of all rasters that are stored somewhere (like RAM or disk).

## Features Defined

- Has a `set_data` method that allows to write pixels to storage

**set\_data** (*array*, *fp=None*, *channels=None*, *interpolation='cv\_area'*, *mask=None*, *\*\*kwargs*)

Write a rectangle of data to the destination raster. Each channel in *array* is written to one channel in *raster* in the same order as described by the *channels* parameter. An optional *mask* may be provided to only write certain pixels of *array*.

If *fp* is not fully within the destination raster, only the overlapping pixels are written. If *fp* is not on the same grid as the destination raster, remapping is automatically performed using the *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to *False* in *Dataset* (default)). When interpolating:

- The nodata values are not interpolated, they are correctly spread to the output.
- At most one pixel may be lost at edges due to interpolation. Provide more context in *array* to compensate this loss.
- The mask parameter is also interpolated.

The alpha bands are currently resampled like any other band, this behavior may change in the future.

This method is not thread-safe.

## Parameters

**array:** `numpy.ndarray` of shape (Y, X) or (Y, X, C) The values to be written

**fp:** Footprint of shape (Y, X) or None If None: write the full source raster If Footprint: write this window to the raster

**channels:** None or int or slice or sequence of int (see *Channels Parameter* below) The channels to be written.

**interpolation:** one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None  
OpenCV method used if interpolation is necessary

**mask:** `numpy array` of shape (Y, X) and dtype *bool* OR inputs accepted by *Footprint.burn\_polygons*

## Channels Parameter

type	value	meaning
NoneType	None (default)	All channels
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels

## Caveat

When using a Raster backed by a driver (like a GDAL driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Raster.

**fill** (*value*, *channels=None*, *\*\*kwargs*)

Fill raster with value.

This method is not thread-safe.

## Parameters

**value:** nbr

**channels:** int or sequence of int (see *Channels Parameter* below) The channels to be written

## Channels Parameter

type	value	meaning
NoneType	None (default)	All channels
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels

## Caveat

When using a Raster backed by a driver (like a GDAL driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Raster.

**class** `buzzard.AEmissary` (<implementation detail>)

Base abstract class defining the common behavior of all sources that are backed by a driver.

## Features Defined

- Has a *driver* (like “GTiff” for GDAL’s geotiff driver)
- Has *open\_options*
- Has a *path* (if the driver supports it)
- Can be deleted (if the driver supports it)

### property driver

Get the driver name, such as ‘GTiff’ or ‘GeoJSON’

### property open\_options

Get the list of options used for opening

### property path

Get the file system path of this source, may be the empty string if not applicable

### property delete

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = ‘r’ The *delete* attribute returns an object that can be both called and used in a with statement

### Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

#### property remove

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = 'r' The *delete* attribute returns an object that can be both called and used in a with statement

### Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

**class** `buzzaard.AEmissaryRaster` (<implementation detail>)

Base abstract class defining the common behavior of all rasters that are backed by a driver.

#### Features Defined

None

**class** `buzzaard.GDALMemRaster` (<implementation detail>)

Concrete class defining the behavior of a GDAL raster using the “MEM” driver.

```
>>> help(Dataset.create_raster)
```

#### Features Defined

None

## 1.2.3 NumpyRaster

**class** `buzzaard.ASource` (<implementation detail>)

Base abstract class defining the common behavior of all sources opened in the Dataset.



## Features Defined

- Has a *stored* spatial reference
- Has a *virtual* spatial reference that is influenced by the Dataset's opening mode
- Can be closed

### property `wkt_stored`

The spatial reference that can be found in the metadata of a source, in wkt format.  
string or None

### property `proj4_stored`

The spatial reference that can be found in the metadata of a source, in proj4 format.  
string or None

### property `wkt_virtual`

The spatial reference considered to be written in the metadata of a source, in wkt format.  
string or None

### property `proj4_virtual`

The spatial reference considered to be written in the metadata of a source, in proj4 format.  
string or None

### method `get_keys()`

Get the list of keys under which this source is registered to in the Dataset

### property `close`

Close a source with a call or a context management. The `close` attribute returns an object that can be both called and used in a with statement

## Examples

```
>>> ds.dem.close()
>>> with ds.dem.close:
    # code...
>>> with ds.create_raster('result.tif', fp, float, 1).close as result:
    # code...
>>> with ds.create_vector('results.shp', 'linestring').close as roofs:
    # code...
```

### method `__del__()`

**class** `buzzard.ASourceRaster` (<implementation detail>)

Base abstract class defining the common behavior of all rasters.

## Features Defined

- Has a *stored* Footprint that defines the location of the raster
- Has a Footprint that is influenced by the Dataset's opening mode
- Has a length that defines how many channels are available
- Has a *channels\_schema* that defines per channel attributes (e.g. nodata)
- Has a *dtype* (like np.float32)
- Has a *get\_data* method that allows to read pixels in their current state to numpy arrays

**property** `fp_stored`

**property** `fp`

**property** `channels_schema`

**property** `dtype`

**property** `nodata`

Accessor for first channel's nodata value

**get\_nodata** (*channel=0*)

Accessor for nodata value

**\_\_len\_\_** ()

Return the number of channels

**get\_data** (*fp=None, channels=None, dst\_nodata=None, interpolation='cv\_area', \*\*kwargs*)

Read a rectangle of data on several channels from the source raster.

If *fp* is not fully within the source raster, the external pixels are set to nodata. If nodata is missing, 0 is used. If *fp* is not on the same grid as the source raster, remapping is performed using *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to False in *Dataset* (default)). When remapping, the nodata values are not interpolated, they are correctly spread to the output.

If *dst\_nodata* is provided, nodata pixels are set to *dst\_nodata*.

**Warning:** The alpha channels are currently resampled like any other channels, this behavior may change in the future. To normalize an *rgba* array after a resampling operation, use this piece of code:

```
>>> arr = np.where(arr[..., -1] == 255, arr, 0)
```

**Warning:** Bands in GDAL are indexed from 1. Channels in buzard are indexed from 0.

## Parameters

**fp: Footprint of shape (Y, X) or None** If None: return the full source raster

If Footprint: return this window from the raster

**channels: None or int or slice or sequence of int (see *Channels Parameter* below)** The channels to be read

**dst\_nodata: nbr or None** nodata value in output array If None and raster.nodata is not None: raster.nodata is used If None and raster.nodata is None: 0 is used

**interpolation: one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None**  
OpenCV method used if interpolation is necessary

## Returns

**array: numpy.ndarray of shape (Y, X) or (Y, X, C)**

- If the *channels* parameter is *-1*, the returned array is of shape (Y, X) when *C=1*, (Y, X, C) otherwise.
- If the *channels* parameter is an integer  $\geq 0$ , the returned array is of shape (Y, X).
- If the *channels* parameter is a sequence or a slice, the returned array is always of shape (Y, X, C), no matter the size of *C*.

(see *Channels Parameter* below)

## Channels Parameter

type	value	meaning	output shape
NoneType	None (default)	All channels	(Y, X) or (Y, X, C)
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels	(Y, X, C)
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>	(Y, X)
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels	(Y, X, C)

**class** buzzard.**AStored** (<implementation detail>)

Base abstract class defining the common behavior of all sources that are stored somewhere (like RAM or disk).

### Features Defined

- Has an opening mode

**property mode**

Open mode, one of {'r', 'w'}

**class** buzzard.**AStoredRaster** (<implementation detail>)

Base abstract class defining the common behavior of all rasters that are stored somewhere (like RAM or disk).

## Features Defined

- Has a `set_data` method that allows to write pixels to storage

**set\_data** (*array*, *fp=None*, *channels=None*, *interpolation='cv\_area'*, *mask=None*, *\*\*kwargs*)

Write a rectangle of data to the destination raster. Each channel in *array* is written to one channel in *raster* in the same order as described by the *channels* parameter. An optional *mask* may be provided to only write certain pixels of *array*.

If *fp* is not fully within the destination raster, only the overlapping pixels are written. If *fp* is not on the same grid as the destination raster, remapping is automatically performed using the *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to *False* in *Dataset* (default)). When interpolating:

- The nodata values are not interpolated, they are correctly spread to the output.
- At most one pixel may be lost at edges due to interpolation. Provide more context in *array* to compensate this loss.
- The mask parameter is also interpolated.

The alpha bands are currently resampled like any other band, this behavior may change in the future.

This method is not thread-safe.

## Parameters

**array:** `numpy.ndarray` of shape (Y, X) or (Y, X, C) The values to be written

**fp:** Footprint of shape (Y, X) or None If None: write the full source raster If Footprint: write this window to the raster

**channels:** None or int or slice or sequence of int (see *Channels Parameter* below) The channels to be written.

**interpolation:** one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None  
OpenCV method used if interpolation is necessary

**mask:** `numpy array` of shape (Y, X) and dtype *bool* OR inputs accepted by *Footprint.burn\_polygons*

## Channels Parameter

type	value	meaning
NoneType	None (default)	All channels
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels

### Caveat

When using a Raster backed by a driver (like a GDAL driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Raster.

**fill** (*value*, *channels=None*, *\*\*kwargs*)

Fill raster with value.

This method is not thread-safe.

### Parameters

**value:** nbr

**channels:** int or sequence of int (see *Channels Parameter* below) The channels to be written

### Channels Parameter

type	value	meaning
NoneType	None (default)	All channels
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels

### Caveat

When using a Raster backed by a driver (like a GDAL driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Raster.

**class** `buzzard.NumpyRaster` (<implementation detail>)

Concrete class defining the behavior of a wrapped numpy array

```
>>> help(Dataset.wrap_numpy_raster)
```

### Features Defined

- Has an *array* property that points to the numpy array provided at construction.

**property** `array`

Returns the Raster's full input data as a Numpy array

## 1.2.4 CachedRasterRecipe

**class** buzzard.**ASource** (<implementation detail>)

Base abstract class defining the common behavior of all sources opened in the Dataset.

### Features Defined

- Has a *stored* spatial reference
- Has a *virtual* spatial reference that is influenced by the Dataset's opening mode
- Can be closed

**property** `wkt_stored`

The spatial reference that can be found in the metadata of a source, in wkt format.

string or None

**property** `proj4_stored`

The spatial reference that can be found in the metadata of a source, in proj4 format.

string or None

**property** `wkt_virtual`

The spatial reference considered to be written in the metadata of a source, in wkt format.

string or None

**property** `proj4_virtual`

The spatial reference considered to be written in the metadata of a source, in proj4 format.

string or None

**get\_keys** ()

Get the list of keys under which this source is registered to in the Dataset

**property** `close`

Close a source with a call or a context management. The `close` attribute returns an object that can be both called and used in a with statement

### Examples

```
>>> ds.dem.close()
>>> with ds.dem.close:
    # code...
>>> with ds.create_raster('result.tif', fp, float, 1).close as result:
    # code...
>>> with ds.create_vector('results.shp', 'linestring').close as roofs:
    # code...
```

**\_\_del\_\_** ()

**class** buzzard.**ASourceRaster** (<implementation detail>)

Base abstract class defining the common behavior of all rasters.

## Features Defined

- Has a *stored* Footprint that defines the location of the raster
- Has a Footprint that is influenced by the Dataset's opening mode
- Has a length that defines how many channels are available
- Has a *channels\_schema* that defines per channel attributes (e.g. nodata)
- Has a *dtype* (like np.float32)
- Has a *get\_data* method that allows to read pixels in their current state to numpy arrays

**property** `fp_stored`

**property** `fp`

**property** `channels_schema`

**property** `dtype`

**property** `nodata`

Accessor for first channel's nodata value

**get\_nodata** (*channel=0*)

Accessor for nodata value

**\_\_len\_\_** ()

Return the number of channels

**get\_data** (*fp=None, channels=None, dst\_nodata=None, interpolation='cv\_area', \*\*kwargs*)

Read a rectangle of data on several channels from the source raster.

If *fp* is not fully within the source raster, the external pixels are set to nodata. If nodata is missing, 0 is used. If *fp* is not on the same grid as the source raster, remapping is performed using *interpolation* algorithm. (It fails if the *allow\_interpolation* parameter is set to False in *Dataset* (default)). When remapping, the nodata values are not interpolated, they are correctly spread to the output.

If *dst\_nodata* is provided, nodata pixels are set to *dst\_nodata*.

**Warning:** The alpha channels are currently resampled like any other channels, this behavior may change in the future. To normalize an *rgba* array after a resampling operation, use this piece of code:

```
>>> arr = np.where(arr[..., -1] == 255, arr, 0)
```

**Warning:** Bands in GDAL are indexed from 1. Channels in buzzard are indexed from 0.

## Parameters

**fp: Footprint of shape (Y, X) or None** If None: return the full source raster

If Footprint: return this window from the raster

**channels: None or int or slice or sequence of int (see *Channels Parameter* below)** The channels to be read

**dst\_nodata: nbr or None** nodata value in output array If None and raster.nodata is not None: raster.nodata is used If None and raster.nodata is None: 0 is used

**interpolation: one of {'cv\_area', 'cv\_nearest', 'cv\_linear', 'cv\_cubic', 'cv\_lanczos4'} or None**  
OpenCV method used if interpolation is necessary

## Returns

**array: numpy.ndarray of shape (Y, X) or (Y, X, C)**

- If the *channels* parameter is *-1*, the returned array is of shape (Y, X) when *C=1*, (Y, X, C) otherwise.
- If the *channels* parameter is an integer  $\geq 0$ , the returned array is of shape (Y, X).
- If the *channels* parameter is a sequence or a slice, the returned array is always of shape (Y, X, C), no matter the size of *C*.

(see *Channels Parameter* below)

## Channels Parameter

type	value	meaning	output shape
NoneType	None (default)	All channels	(Y, X) or (Y, X, C)
slice	slice(None), slice(1), slice(0, 2), slice(2, 0, -1)	Those channels	(Y, X, C)
int	0, 1, 2, -1, -2, -3	Channel <i>idx</i>	(Y, X)
(int, ...)	[0], [1], [2], [-1], [-2], [-3], [0, 1], [-1, 2, 1]	Those channels	(Y, X, C)

**class** `buzzard.AAAsyncRaster` (<*implementation detail*>)

Base abstract class defining the common behavior of all rasters that are managed by the Dataset's scheduler.

## Features Defined

- Has a *queue\_data*, a low level method that can be used to query several arrays at once.
- Has an *iter\_data*, a higher level wrapper of *queue\_data*.

**queue\_data** (*fps*, *channels=None*, *dst\_nodata=None*, *interpolation='cv\_area'*, *max\_queue\_size=5*, *\*\*kwargs*)

Read several rectangles of data on several channels from the source raster.

Using *queue\_data* instead of multiple calls to *get\_data* allows more parallelism. The *fps* parameter should contain a sequence of *Footprint* that will be mapped to *numpy.ndarray*. The first ones will be computed with a higher priority than the later ones.

Calling this method sends an asynchronous message to the Dataset's scheduler with the input parameters and a queue. On the input side of the queue, the scheduler will call the *put* method with each array



requested. On the output side of the queue, the *get* method should be called to retrieve the requested arrays.

The output queue will be created with a max queue size of *max\_queue\_size*, the scheduler will be careful to prepare only the arrays that can fit in the output queue. Thanks to this feature: backpressure can be entirely avoided.

If you wish to cancel your request, loose the reference to the queue and the scheduler will gracefully cancel the query.

In general you should use the *iter\_data* method instead of the *queue\_data* one, it is much safer to use. However you will need to pass the *queue\_data* method of a raster, to create another raster (a recipe) that depends on the first raster.

see rasters' *get\_data* documentation, it shares most of the concepts

### Parameters

**fps: sequence of Footprint** The Footprints at which the raster should be sampled.

**channels:** see *get\_data* method

**dst\_nodata:** see *get\_data* method

**interpolation:** see *get\_data* method

**max\_queue\_size: int** Maximum number of arrays to prepare in advance in the underlying queue.

### Returns

**queue: queue.Queue of ndarray** The arrays are put into the queue in the same order as in the *fps* parameter.

**iter\_data** (*fps*, *channels=None*, *dst\_nodata=None*, *interpolation='cv\_area'*, *max\_queue\_size=5*, *\*\*kwargs*)

Read several rectangles of data on several channels from the source raster.

The *iter\_data* method is a higher level wrapper around the *queue\_data* method. It returns a python generator and while waiting for data, it periodically probes the Dataset's scheduler to reraise an exception if it crashed.

If you wish to cancel your request, loose the reference to the iterable and the scheduler will gracefully cancel the query.

see rasters' *get\_data* documentation, it shares most of the concepts see *queue\_data* documentation, it is called from within the *iter\_data* method

### Parameters

**fps: sequence of Footprint** The Footprints at which the raster should be sampled.

**channels:** see *get\_data* method

**dst\_nodata:** see *get\_data* method

**interpolation:** see *get\_data* method

**max\_queue\_size: int** Maximum number of arrays to prepare in advance in the underlying queue.

## Returns

**iterable: iterable of ndarray** The arrays are yielded into the generator in the same order as in the *fps* parameter.

**class** `buzzard.ARasterRecipe` (<implementation detail>)

Base abstract class defining the common behavior of all rasters that compute data on the fly through the Dataset's scheduler.

## Features Defined

- Has a *primitives* property, a dict that lists the primitive rasters declared at construction.

**property** `primitives`

dict of primitive name to Source, deduced from the *queue\_data\_per\_primitive* provided at construction.

**class** `buzzard.CachedRasterRecipe` (<implementation detail>)

Concrete class defining the behavior of a raster computed on the fly and fills a cache to avoid subsequent computations.

```
>>> help(Dataset.create_cached_raster_recipe)
```

**property** `cache_tiles`

Cache tiles provided or created at construction

**property** `cache_dir`

Cache directory path provided at construction

## 1.2.5 GDALFileVector

**class** `buzzard.ASource` (<implementation detail>)

Base abstract class defining the common behavior of all sources opened in the Dataset.

## Features Defined

- Has a *stored* spatial reference
- Has a *virtual* spatial reference that is influenced by the Dataset's opening mode
- Can be closed

**property** `wkt_stored`

The spatial reference that can be found in the metadata of a source, in wkt format.

string or None

**property** `proj4_stored`

The spatial reference that can be found in the metadata of a source, in proj4 format.

string or None

**property** `wkt_virtual`

The spatial reference considered to be written in the metadata of a source, in wkt format.

string or None

**property proj4\_virtual**

The spatial reference considered to be written in the metadata of a source, in proj4 format.

string or None

**get\_keys ()**

Get the list of keys under which this source is registered to in the Dataset

**property close**

Close a source with a call or a context management. The *close* attribute returns an object that can be both called and used in a with statement

**Examples**

```
>>> ds.dem.close()
>>> with ds.dem.close:
    # code...
>>> with ds.create_raster('result.tif', fp, float, 1).close as result:
    # code...
>>> with ds.create_vector('results.shp', 'linestring').close as roofs:
    # code...
```

**\_\_del\_\_ ()**

**class** buzzard.ASourceVector (<implementation detail>)

Base abstract class defining the common behavior of all vectors.

**Features Defined**

- Has a *type* that defines the type of geometry (like “Polygon”)
- Has *fields* that define the type of informations that is paired with each geometries
- Has a *stored* extent that allows to retrieve the current extent of all the geometries
- Has a length that indicates how many geometries this source contains.
- Has several read functions (like *iter\_data*) to retrieve geometries in their current state to shapely objects

**property type**

Geometry type

**property fields**

Fields definition

**property extent**

Get the vector’s extent in work spatial reference. (*x* then *y*)

### Example

```
>>> minx, maxx, miny, maxy = ds.roofs.extent
```

#### **property extent\_stored**

Get the vector's extent in stored spatial reference. (minx, miny, maxx, maxy)

#### **property bounds**

Get the vector's bounds in work spatial reference. (*min* then *max*)

### Example

```
>>> minx, miny, maxx, maxy = ds.roofs.extent
```

#### **property bounds\_stored**

Get the vector's bounds in stored spatial reference. (*min* then *max*)

#### **\_\_len\_\_()**

Return the number of features in vector

#### **iter\_data** (*fields=None, geom\_type='shapely', mask=None, clip=False, slicing=slice(0, None, 1)*)

Create an iterator over vector's features

### Parameters

**fields:** **None or string or -1 or sequence of string/int** Which fields to include in iteration

- if None, empty sequence or empty string: No fields included
- if -1: All fields included
- if string: Name of fields to include (separated by comma or space)
- if sequence: List of indices / names to include

**geom\_type:** {'shapely', 'coordinates'} Returned geometry type

**mask:** **None or Footprint or shapely geometry or (nbr, nbr, nbr, nbr)** Add a spatial filter to iteration, only geometries not disjoint with mask will be included.

- if None: No spatial filter
- if Footprint or shapely polygon: Polygon
- if (nbr, nbr, nbr, nbr): Extent (minx, maxx, miny, maxy)

**clip:** **bool** Returns intersection of geometries and mask. Caveat: A clipped geometry might not be of the same type as the original geometry. e.g: polygon might be clipped to might be converted to one of those:

- polygon
- line
- point
- multipolygon
- multiline
- multipoint

- geometrycollection

**slicing: slice** Slice of the iteration to return. It is applied after spatial filtering

## Yields

**feature: geometry or (geometry,) or (geometry, \*fields)**

- If *geom\_type* is 'shapely': geometry is a *shapely geometry*.
- If *geom\_type* is *coordinates*: geometry is a *nested lists of numpy arrays*.
- If *fields* is not a sequence: *feature* is *geometry* or *(geometry, \*fields)*, depending on the number of fields to yield.
- If *fields* is a sequence or a string: *feature* is *(geometry,)* or *(geometry, \*fields)*. Use *fields=[-1]* to get a monad containing all fields.

## Examples

```
>>> for polygon, volume, stock_type in ds.stocks.iter_data('volume,type'):
    print('area: {}m**2, volume: {}m**3'.format(polygon.area, volume))
```

```
>>> for polygon, in ds.stocks.iter_data([]):
    print('area: {}m**2'.format(polygon.area))
```

```
>>> for polygon in ds.stocks.iter_data():
    print('area: {}m**2'.format(polygon.area))
```

**get\_data** (*index, fields=-1, geom\_type='shapely', mask=None, clip=False*)

Fetch a single feature in vector. See `ASourceVector.iter_data`

**iter\_geojson** (*mask=None, clip=False, slicing=slice(0, None, 1)*)

Create an iterator over vector's features

## Parameters

**mask: None or Footprint or shapely geometry or (nbr, nbr, nbr, nbr)** Add a spatial filter to iteration, only geometries not disjoint with mask will be included.

- if None: No spatial filter
- if Footprint or shapely polygon: Polygon
- if (nbr, nbr, nbr, nbr): Extent (minx, maxx, miny, maxy)

**clip: bool** Returns intersection of geometries and mask. Caveat: A clipped geometry might not be of the same type as the original geometry. e.g: polygon might be clipped to might be converted to one of those:

- polygon
- line
- point
- multipolygon
- multiline

- multipoint
- geometrycollection

**slicing:** `slice` Slice of the iteration to return. It is applied after spatial filtering

## Returns

iterable of geojson feature (dict)

## Example

```
>>> for geojson in ds.stocks.iter_geojson():
    print('exterior-point-count:{}, volume:{}'.format(
        len(geojson['geometry']['coordinates'][0]),
        geojson['properties']['volume']
    ))
```

**get\_geojson** (*index*, *mask=None*, *clip=False*)

Fetch a single feature in vector. See `ASourceVector.iter_geojson`

**extent\_origin**

Descriptor object to manage deprecation

**class** `buzzard.AStored` (<implementation detail>)

Base abstract class defining the common behavior of all sources that are stored somewhere (like RAM or disk).

## Features Defined

- Has an opening mode

**property** `mode`

Open mode, one of {'r', 'w'}

**class** `buzzard.AStoredVector` (<implementation detail>)

Base abstract class defining the common behavior of all vectors that are stored somewhere (like RAM or disk).

## Features Defined

- Has an `insert_data` method that allows to write geometries to storage

**insert\_data** (*geom*, *fields=()*, *index=-1*)

Insert a feature in vector.

This method is not thread-safe.

## Parameters

**geom:** `shapely.base.BaseGeometry` or nested sequence of coordinates

**fields:** **sequence or dict** Feature's fields, missing or None fields are defaulted.

- if empty sequence: Keep all fields defaulted
- if sequence of length `len(self.fields)`: Fields to be set, same order as `self.fields`
- if dict: Mapping of fields to be set

**index:** **int**

- if -1: append feature
- else: insert feature at index (if applicable)

## Example

```
>>> poly = shapely.geometry.box(10, 10, 42, 43)
>>> fields = {'volume': 42.24}
>>> ds.stocks.insert_data(poly, fields)
```

## Caveat

When using a Vector backed by a driver (like an OGR driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Vector.

**class** `buzzard.AEmissary` (*<implementation detail>*)

Base abstract class defining the common behavior of all sources that are backed by a driver.

## Features Defined

- Has a *driver* (like “GTiff” for GDAL’s geotiff driver)
- Has *open\_options*
- Has a *path* (if the driver supports it)
- Can be deleted (if the driver supports it)

**property driver**

Get the driver name, such as ‘GTiff’ or ‘GeoJSON’

**property open\_options**

Get the list of options used for opening

**property path**

Get the file system path of this source, may be the empty string if not applicable

**property delete**

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = ‘r’ The *delete* attribute returns an object that can be both called and used in a with statement

### Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

#### property `remove`

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = 'r' The *delete* attribute returns an object that can be both called and used in a with statement

### Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

**class** `buzzaard.AEmissaryVector` (<implementation detail>)

Base abstract class defining the common behavior of all vectors that are backed by a driver.

#### Features Defined

- Has a *layer* (if the driver supports it)

#### property `layer`

**class** `buzzaard.APooledEmissary` (<implementation detail>)

Base abstract class defining the common behavior of all sources that can deactivate and reactivate their underlying driver at will.

This is useful to balance the number of active file descriptors. This is useful to perform concurrent reads if the driver does not support it.

#### Features Defined

- An *activate* method to manually open the driver (Mostly useless feature since opening is automatic if necessary)
- A *deactivate* method to close the driver (Useful to flush data to disk)
- An *active\_count* property
- An *active* property

#### `activate` ()

Make sure that at least one driver object is active for this Raster/Vector

#### `deactivate` ()

Collect all active driver object for this Raster/Vector. If a driver object is currently being used, will raise an exception.



**property active\_count**

Count how many driver objects are currently active for this Raster/Vector

**property active**

Is there any driver object currently active for this Raster/Vector

**class** `buzzard.APooledEmissaryVector` (*<implementation detail>*)

Base abstract class defining the common behavior of all vectors that can deactivate and reactivate their underlying driver at will.

**Features Defined**

None

**class** `buzzard.GDALFileVector` (*<implementation detail>*)

Concrete class defining the behavior of a GDAL vector using a file

```
>>> help(Dataset.open_vector)
>>> help(Dataset.create_vector)
```

**Features Defined**

None

## 1.2.6 GDALMemoryVector

**class** `buzzard.ASource` (*<implementation detail>*)

Base abstract class defining the common behavior of all sources opened in the Dataset.

**Features Defined**

- Has a *stored* spatial reference
- Has a *virtual* spatial reference that is influenced by the Dataset's opening mode
- Can be closed

**property wkt\_stored**

The spatial reference that can be found in the metadata of a source, in wkt format.

string or None

**property proj4\_stored**

The spatial reference that can be found in the metadata of a source, in proj4 format.

string or None

**property wkt\_virtual**

The spatial reference considered to be written in the metadata of a source, in wkt format.

string or None

**property proj4\_virtual**

The spatial reference considered to be written in the metadata of a source, in proj4 format.

string or None

**get\_keys()**

Get the list of keys under which this source is registered to in the Dataset

**property close**

Close a source with a call or a context management. The *close* attribute returns an object that can be both called and used in a with statement

**Examples**

```
>>> ds.dem.close()
>>> with ds.dem.close:
    # code...
>>> with ds.acreate_raster('result.tif', fp, float, 1).close as result:
    # code...
>>> with ds.acreate_vector('results.shp', 'linestring').close as roofs:
    # code...
```

**\_\_del\_\_()**

**class** buzzard.ASourceVector (<implementation detail>)

Base abstract class defining the common behavior of all vectors.

**Features Defined**

- Has a *type* that defines the type of geometry (like “Polygon”)
- Has *fields* that define the type of informations that is paired with each geometries
- Has a *stored* extent that allows to retrieve the current extent of all the geometries
- Has a length that indicates how many geometries this source contains.
- Has several read functions (like *iter\_data*) to retrieve geometries in their current state to shapely objects

**property type**

Geometry type

**property fields**

Fields definition

**property extent**

Get the vector’s extent in work spatial reference. (*x* then *y*)

**Example**

```
>>> minx, maxx, miny, maxy = ds.roofs.extent
```

**property extent\_stored**

Get the vector’s extent in stored spatial reference. (minx, miny, maxx, maxy)

**property bounds**

Get the vector’s bounds in work spatial reference. (*min* then *max*)

## Example

```
>>> minx, miny, maxx, maxy = ds.roofs.extent
```

### **property** `bounds_stored`

Get the vector's bounds in stored spatial reference. (*min* then *max*)

### **\_\_len\_\_** ()

Return the number of features in vector

### **iter\_data** (*fields=None, geom\_type='shapely', mask=None, clip=False, slicing=slice(0, None, 1)*)

Create an iterator over vector's features

## Parameters

**fields:** **None or string or -1 or sequence of string/int** Which fields to include in iteration

- if None, empty sequence or empty string: No fields included
- if -1: All fields included
- if string: Name of fields to include (separated by comma or space)
- if sequence: List of indices / names to include

**geom\_type:** {'shapely', 'coordinates'} Returned geometry type

**mask:** **None or Footprint or shapely geometry or (nbr, nbr, nbr, nbr)** Add a spatial filter to iteration, only geometries not disjoint with mask will be included.

- if None: No spatial filter
- if Footprint or shapely polygon: Polygon
- if (nbr, nbr, nbr, nbr): Extent (minx, maxx, miny, maxy)

**clip:** **bool** Returns intersection of geometries and mask. Caveat: A clipped geometry might not be of the same type as the original geometry. e.g: polygon might be clipped to might be converted to one of those:

- polygon
- line
- point
- multipolygon
- multiline
- multipoint
- geometrycollection

**slicing:** **slice** Slice of the iteration to return. It is applied after spatial filtering

## Yields

### feature: geometry or (geometry,) or (geometry, \*fields)

- If *geom\_type* is 'shapely': geometry is a *shapely geometry*.
- If *geom\_type* is *coordinates*: geometry is a *nested lists of numpy arrays*.
- If *fields* is not a sequence: *feature* is *geometry* or *(geometry, \*fields)*, depending on the number of fields to yield.
- If *fields* is a sequence or a string: *feature* is *(geometry,)* or *(geometry, \*fields)*. Use *fields=[-1]* to get a monad containing all fields.

## Examples

```
>>> for polygon, volume, stock_type in ds.stocks.iter_data('volume,type'):  
    print('area: {}m**2, volume: {}m**3'.format(polygon.area, volume))
```

```
>>> for polygon, in ds.stocks.iter_data([]):  
    print('area: {}m**2'.format(polygon.area))
```

```
>>> for polygon in ds.stocks.iter_data():  
    print('area: {}m**2'.format(polygon.area))
```

**get\_data** (*index, fields=-1, geom\_type='shapely', mask=None, clip=False*)

Fetch a single feature in vector. See `ASourceVector.iter_data`

**iter\_geojson** (*mask=None, clip=False, slicing=slice(0, None, 1)*)

Create an iterator over vector's features

## Parameters

**mask:** **None** or **Footprint** or **shapely geometry** or **(nbr, nbr, nbr, nbr)** Add a spatial filter to iteration, only geometries not disjoint with mask will be included.

- if **None**: No spatial filter
- if **Footprint** or **shapely polygon**: **Polygon**
- if **(nbr, nbr, nbr, nbr)**: **Extent** (**minx, maxx, miny, maxy**)

**clip:** **bool** Returns intersection of geometries and mask. Caveat: A clipped geometry might not be of the same type as the original geometry. e.g: polygon might be clipped to might be converted to one of those:

- **polygon**
- **line**
- **point**
- **multipolygon**
- **multiline**
- **multipoint**
- **geometrycollection**

**slicing:** **slice** Slice of the iteration to return. It is applied after spatial filtering

## Returns

iterable of geojson feature (dict)

## Example

```
>>> for geojson in ds.stocks.iter_geojson():
    print('exterior-point-count:{}, volume:{}.format(
        len(geojson['geometry']['coordinates'][0]),
        geojson['properties']['volume']
    ))
```

**get\_geojson** (*index*, *mask=None*, *clip=False*)

Fetch a single feature in vector. See ASourceVector.iter\_geojson

**extent\_origin**

Descriptor object to manage deprecation

**class** buzzard.AStored (<implementation detail>)

Base abstract class defining the common behavior of all sources that are stored somewhere (like RAM or disk).

## Features Defined

- Has an opening mode

**property mode**

Open mode, one of {'r', 'w'}

**class** buzzard.AStoredVector (<implementation detail>)

Base abstract class defining the common behavior of all vectors that are stored somewhere (like RAM or disk).

## Features Defined

- Has an *insert\_data* method that allows to write geometries to storage

**insert\_data** (*geom*, *fields=()*, *index=-1*)

Insert a feature in vector.

This method is not thread-safe.

## Parameters

**geom**: shapely.base.BaseGeometry or nested sequence of coordinates

**fields**: sequence or dict Feature's fields, missing or None fields are defaulted.

- if empty sequence: Keep all fields defaulted
- if sequence of length len(self.fields): Fields to be set, same order as self.fields
- if dict: Mapping of fields to be set

**index**: int

- if -1: append feature
- else: insert feature at index (if applicable)

### Example

```
>>> poly = shapely.geometry.box(10, 10, 42, 43)
>>> fields = {'volume': 42.24}
>>> ds.stocks.insert_data(poly, fields)
```

### Caveat

When using a Vector backed by a driver (like an OGR driver), the data might be flushed to disk only after the garbage collection of the driver object. To be absolutely sure that the driver cache is flushed to disk, call `.close` or `.deactivate` on this Vector.

**class** `buzzaard.AEmissary` (<implementation detail>)

Base abstract class defining the common behavior of all sources that are backed by a driver.

### Features Defined

- Has a *driver* (like “GTiff” for GDAL’s geotiff driver)
- Has *open\_options*
- Has a *path* (if the driver supports it)
- Can be deleted (if the driver supports it)

#### **property driver**

Get the driver name, such as ‘GTiff’ or ‘GeoJSON’

#### **property open\_options**

Get the list of options used for opening

#### **property path**

Get the file system path of this source, may be the empty string if not applicable

#### **property delete**

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = ‘r’ The *delete* attribute returns an object that can be both called and used in a with statement

### Example

```
>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...
```

#### **property remove**

Delete a source with a call or a context management. May raise an exception if not applicable or if *mode* = ‘r’ The *delete* attribute returns an object that can be both called and used in a with statement

## Example

```

>>> ds.dem.delete()
>>> with ds.dem.delete:
    # code...
>>> with ds.create_raster('/tmp/tmp.tif', fp, float, 1).delete as tmp:
    # code...
>>> with ds.create_vector('/tmp/tmp.shp', 'polygon').delete as tmp:
    # code...

```

**class** `buzzard.AEmissaryVector` (<implementation detail>)

Base abstract class defining the common behavior of all vectors that are backed by a driver.

### Features Defined

- Has a *layer* (if the driver supports it)

#### property `layer`

**class** `buzzard.GDALMemoryVector` (<implementation detail>)

Concrete class defining the behavior of a GDAL raster using the “Memory” driver

```

>>> help(Dataset.create_vector)

```

### Features Defined

None

## 1.3 Footprint

**class** `buzzard.Footprint` (\*\*kwargs)

Immutable object representing the location and size of a spatially localized raster. All methods are thread-safe.

The `Footprint` class:

- is a toolbox class designed to position a rectangle in both image space and geometry space,
- can be seen as a `shapely.geometry.Polygon` rectangle that also defines a grid of pixels,
- its main purpose is to simplify the manipulation of windows in rasters,
- has many accessors,
- has many algorithms,
- is a constant object,
- is designed to work with any rectangle in space (like non north-up/west-left rasters),
- is independent from projections, units and files,
- uses `affine` library internally for conversions (<https://github.com/sgillies/affine>).

**Warning:** This class being complex and full python, the constructor is too slow for certain use cases (~0.5ms).

Method category		Method names
Footprint construction	from scratch	<code>__init__</code> , <code>of_extent</code>
	from <code>Footprint</code>	<code>__and__</code> , <code>intersection</code> , <code>erode</code> , <code>dilate</code> , ...
Conversion		<code>extent</code> , <code>coords</code> , <code>geom</code> , <code>__geo_interface__</code>
Accessors	Spatial - Size and vectors	<code>size</code> , <code>width</code> , <code>height</code> , <code>diagvec</code> , ...
	Spatial - Coordinates	<code>tl</code> , <code>bl</code> , <code>br</code> , <code>tr</code> , ...
	Spatial - Misc	<code>area</code> , <code>length</code> , <code>semiminoraxis</code> , ...
	Raster - Size	<code>rsize</code> , <code>rwidth</code> , <code>rheight</code> , ...
	Raster - Indices	<code>rtl</code> , <code>rbl</code> , <code>rbr</code> , <code>ttr</code> , ...
	Raster - Misc	<code>rarea</code> , <code>rlength</code> , <code>rsemiminoraxis</code> , ...
Affine transformations		<code>pxsize</code> , <code>pxvec</code> , <code>angle</code> , ...
Binary predicates		<code>__eq__</code> , ...
Numpy		<code>shape</code> , <code>meshgrid_raster</code> , <code>meshgrid_spatial</code> , <code>slice_in</code> , ...
Coordinates conversions		<code>spatial_to_raster</code> , <code>raster_to_spatial</code>
Geometry / Raster conversions		<code>find_polygons</code> , <code>burn_polygons</code> , ...
Tiling		<code>tile</code> , <code>tile_count</code> , <code>tile_occurrence</code>
Serialization		<code>__str__</code> , ...

### Informations on geo transforms (gt) and affine matrices

- <http://www.perrygeo.com/python-affine-transforms.html>
- <https://pypi.python.org/pypi/affine/1.0>

GDAL ordering:

c	a	b	f	d	e
tlx	width of a pixel	row rotation	tly	column rotation	height of a pixel

```
>>> c, a, b, f, d, e = fp.gt
>>> tlx, dx, rx, tly, ry, dy = fp.gt
```

Matrix ordering:

a	b	c	d	e	f
width of a pixel	row rotation	tlx	column rotation	height of a pixel	tly

```
>>> a, b, c, d, e, f = fp.aff6
>>> dx, rx, tlx, ry, dy, tly = fp.aff6
```

There are only two ways to construct a Footprint, but several high level constructors exist, such as `.intersection`.

Usage 1

```
>>> buzz.Footprint(tl=(0, 10), size=(10, 10), rsize=(100, 100))
```

Usage 2

```
>>> buzz.Footprint(gt=(0, .1, 0, 10, 0, -.1), rsize=(100, 100))
```



## Parameters

**tl:** (**nbr, nbr**) raster spatial top left coordinates

**gt:** (**nbr, nbr, nbr, nbr, nbr, nbr**) geotransforms with GDAL ordering

**size:** (**nbr, nbr**) Size of Footprint in space (unsigned)

**rsize:** (**int, int**) Size of raster in pixel (unsigned integers)

**\_\_and\_\_** (*other*)

Returns Footprint.intersection

**classmethod of\_extent** (*extent, scale*)

Create a Footprint from a rectangle extent and a scale

## Parameters

**extent:** (**nbr, nbr, nbr, nbr**) Spatial coordinates of (minx, maxx, miny, maxy) defining a rectangle

**scale:** **nbr or (nbr, nbr)** Resolution of output Footprint:

- if nbr: resolution = [a, -a]
- if (nbr, nbr): resolution [a, b]

**clip** (*startx, starty, endx, endy*)

Construct a new Footprint by clipping self using pixel indices

To clip using coordinates see *Footprint.intersection*.

## Parameters

**startx:** **int or None** Same rules as regular python slicing

**starty:** **int or None** Same rules as regular python slicing

**endx:** **int or None** Same rules as regular python slicing

**endy:** **int or None** Same rules as regular python slicing

## Returns

**fp:** **Footprint** The new clipped Footprint

**erode** (*self, inward\_count, /*)

erode(self, inward\_count\_x, inward\_count\_y, /) erode(self, inward\_count\_left, inward\_count\_right, inward\_count\_top, inward\_count\_bottom, /)

Erode self's edges by the given pixel count to construct a new Footprint.

A negative erosion is a dilation.

### Parameters

**\*args: int or (int, int) or (int, int, int, int)** When *int*, erode all 4 directions by that much pixels When (*int, int*), erode x and y by a different number of pixel When (*int, int, int, int*), erode all 4 directions with a different number of pixel

### Returns

Footprint

**dilate** (*self, outward\_count, /*)

dilate(*self, outward\_count\_x, outward\_count\_y, /*) dilate(*self, outward\_count\_left, outward\_count\_right, outward\_count\_top, outward\_count\_bottom, /*)

Dilate self's edges by the given pixel count to construct a new Footprint.

A negative dilation is an erosion.

### Parameters

**\*args: int or (int, int) or (int, int, int, int)** When *int*, dilate all 4 directions by that much pixels When (*int, int*), dilate x and y by a different number of pixel When (*int, int, int, int*), dilate all 4 directions with a different number of pixel

### Returns

Footprint

**intersection** (*self, \*objects, scale='self', rotation='auto', alignment='auto', homogeneous=False*)

Construct a Footprint bounding the intersection of geometric objects, self being one of the of input geometry. Inputs' intersection is always within output Footprint.

### Parameters

**\*objects: \*object** Any object with a `__geo_interface__` attribute defining a geometry, like a Footprint or a shapely object.

**scale: one of {'self', 'highest', 'lowest'} or (nbr, nbr) or nbr** 'self': Output Footprint's resolution is the same as self 'highest': Output Footprint's resolution is the highest one among the input Footprints 'lowest': Output Footprint's resolution is the lowest one among the input Footprints (nbr, nbr): Signed pixel size, aka scale nbr: Signed pixel width. Signed pixel height is assumed to be -width

**rotation: one of {'auto', 'fit'} or nbr**

'auto' If *scale* designate a Footprint object, its rotation is chosen Else, self's rotation is chosen

'fit' Output Footprint is the rotated minimum bounding rectangle

**nbr** Angle in degree

**alignment: {'auto', 'tl', (nbr, nbr)}**

'auto'

If *scale* and *rotation* designate the same Footprint object, its **alignment** is chosen

Else, 'tl' alignment is chosen

**‘tl’**: Output Footprint’s alignment is the top left most point of the bounding rectangle of the intersection

**(nbr, nbr)**: Coordinate of a point that lie on the grid. This point can be anywhere in space.

**homogeneous**: **bool** False: No effect True: Raise an exception if all input Footprints do not lie on the same grid as self.

## Returns

Footprint

**move** (*tl*, *tr=None*, *br=None*, *round\_coordinates=False*)

Create a copy of self moved by an Affine transformation by providing new points. *rsize* is always conserved

## Usage cases

tl	tr	br	Affine transformations possible
coord	None	None	Translation
coord	coord	None	Translation, Rotation, Scale x and y uniformly with positive real
coord	coord	coord	Translation, Rotation, Scale x and y independently with reals

## Parameters

**tl**: (**nbr, nbr**) New top left coordinates

**tr**: (**nbr, nbr**) New top right coordinates

**br**: (**nbr, nbr**) New bottom right coordinates

**round\_coordinates**: **bool** Round the input coordinates with respect to *buzz.env.significant*, so that the output *Footprint* is as much similar as possible as the input *Footprint* regarding those properties: - angle - pxsize - pxsize<sub>x</sub> / pxsize<sub>y</sub>

This option helps a lot if the input coordinates suffered from floating point precision loss since it will cancel the noise in the resulting transformation matrix.

**Warning:** Only work when *tr* and *br* are both provided

## Returns

Footprint

**property extent**

Get the Footprint’s extent (x then y)

### Example

```
>>> minx, maxx, miny, maxy = fp.extent
>>> plt.imshow(arr, extent=fp.extent)
```

fp.extent from fp.bounds using numpy fancy indexing

```
>>> minx, maxx, miny, maxy = fp.bounds[[0, 2, 1, 3]]
```

### property bounds

Get the Footprint's bounds (*min* then *max*)

### Example

```
>>> minx, miny, maxx, maxy = fp.bounds
```

fp.bounds from fp.extent using numpy fancy indexing

```
>>> minx, miny, maxx, maxy = fp.extent[[0, 2, 1, 3]]
```

### property coords

Get corners coordinates

### Example

```
>>> tl, bl, br, tr = fp.coords
```

### property poly

Convert self to shapely.geometry.Polygon

### property \_\_geo\_interface\_\_

### property size

(llraster left - raster rightll, llraster top - raster bottomll)

**Type** Spatial distances

### property sizex

llraster left - raster rightll

**Type** Spatial distance

### property sizey

llraster top - raster bottomll

**Type** Spatial distance

### property width

llraster left - raster rightll, alias for sizex

**Type** Spatial distance

### property height

llraster top - raster bottomll, alias for sizey

**Type** Spatial distance

### property w

llraster left - raster rightll, alias for sizex

**Type** Spatial distance

**property h**

llraster top - raster bottomll, alias for sizey

**Type** Spatial distance

**property lrvec**

(raster right - raster left)

**Type** Spatial vector

**property tbvec**

(raster bottom - raster top)

**Type** Spatial vector

**property diagvec**

(raster bottom right - raster top left)

**Type** Spatial vector

**property tl**

raster top left (x, y)

**Type** Spatial coordinates

**property tlx**

raster top left (x)

**Type** Spatial coordinate

**property tly**

raster top left (y)

**Type** Spatial coordinate

**property bl**

raster bottom left (x, y)

**Type** Spatial coordinates

**property blx**

raster bottom left (x)

**Type** Spatial coordinate

**property bly**

raster bottom left (y)

**Type** Spatial coordinate

**property br**

raster bottom right (x, y)

**Type** Spatial coordinates

**property brx**

raster bottom right (x)

**Type** Spatial coordinate

**property bry**

raster bottom right (y)

**Type** Spatial coordinate

**property tr**

raster top right (x, y)

**Type** Spatial coordinates

**property trx**

raster top right (x)

**Type** Spatial coordinate

**property try\_**

raster top right (y) Don't forget the trailing underscore

**Type** Spatial coordinate

**property t**

raster top center (x, y)

**Type** Spatial coordinates

**property tx**

raster top center (x)

**Type** Spatial coordinate

**property ty**

raster top center (y)

**Type** Spatial coordinate

**property l**

raster center left (x, y)

**Type** Spatial coordinates

**property lx**

raster center left (x)

**Type** Spatial coordinate

**property ly**

raster center left (y)

**Type** Spatial coordinate

**property b**

raster bottom center (x, y)

**Type** Spatial coordinates

**property bx**

raster bottom center (x)

**Type** Spatial coordinate

**property by**

raster bottom center (y)

**Type** Spatial coordinate

**property r**

raster center right (x, y)

**Type** Spatial coordinates

**property rx**

raster center right (x)

**Type** Spatial coordinate

**property ry**

raster center right (y)

**Type** Spatial coordinate

**property c**

raster center (x, y)

**Type** Spatial coordinates

**property cx**

raster center (x)

**Type** Spatial coordinate

**property cy**

raster center (y)

**Type** Spatial coordinate

**property semiminoraxis**

half-size of the smaller side

**Type** Spatial distance

**property semimajoraxis**

half-size of the bigger side

**Type** Spatial distance

**property area**

pixel count

**Type** Area

**property length**

circumference of the outer ring

**Type** Length

**property rsize**

(pixel per line, pixel per column)

**Type** Pixel quantities

**property rsize<sub>x</sub>**

pixel per line

**Type** Pixel quantity

**property rsize<sub>y</sub>**

pixel per column

**Type** Pixel quantity

**property rwidth**

pixel per line, alias for rsize<sub>x</sub>

**Type** Pixel quantity

**property rheight**

pixel per column, alias for rsize<sub>y</sub>

**Type** Pixel quantity

**property rw**

pixel per line, alias for rsizex

**Type** Pixel quantity

**property rh**

pixel per column, alias for rsizey

**Type** Pixel quantity

**property rtl**

raster top left pixel (x=0, y=0)

**Type** Indices

**property rtlx**

raster top left pixel (x=0)

**Type** Index

**property rtly**

raster top left pixel (y=0)

**Type** Index

**property rbl**

raster bottom left pixel (x=0, y)

**Type** Indices

**property rblx**

raster bottom left pixel (x=0)

**Type** Index

**property rbly**

raster bottom left pixel (y)

**Type** Index

**property rbr**

raster bottom right pixel (x, y)

**Type** Indices

**property rbrx**

raster bottom right pixel (x)

**Type** Index

**property rbry**

raster bottom right pixel (y)

**Type** Index

**property rtr**

raster top right pixel (x, y=0)

**Type** Indices

**property rtrx**

raster top right pixel (x)

**Type** Index

**property rtry**

raster top right pixel (y=0)



**Type Index**

**property rt**

raster top center pixel (x truncated, y=0)

**Type Indices**

**property rtx**

raster top center pixel (x truncated)

**Type Index**

**property rty**

raster top center pixel (y=0)

**Type Index**

**property rl**

raster center left pixel (x=0, y truncated)

**Type Indices**

**property rlx**

raster center left pixel (x=0)

**Type Index**

**property rly**

raster center left pixel (y truncated)

**Type Index**

**property rb**

raster bottom center pixel (x truncated, y)

**Type Indices**

**property rbx**

raster bottom center pixel (x truncated)

**Type Index**

**property rby**

raster bottom center pixel (y)

**Type Index**

**property rr**

raster center right pixel (x, y truncated)

**Type Indices**

**property rrx**

raster center right pixel (x)

**Type Index**

**property rry**

raster center right pixel (y truncated)

**Type Index**

**property rc**

raster center pixel (x truncated, y truncated)

**Type Indices**

**property rcx**

raster center pixel (x truncated)

**Type** Index

**property rcy**

raster center pixel (y truncated)

**Type** Index

**property rsemiminoraxis**

half pixel count (truncated) of the smaller side

**Type** Pixel quantity

**property rsemimajoraxis**

half pixel count (truncated) of the bigger side

**Type** Pixel quantity

**property rarea**

pixel count

**Type** Pixel quantity

**property rlength**

pixel count in the outer ring

**Type** Pixel quantity

**property gt**

First 6 numbers of the affine transformation matrix, GDAL ordering

**property aff33**

The affine transformation matrix

**property aff23**

Top two rows of the affine transformation matrix

**property aff6**

First 6 numbers of the affine transformation matrix, left-right/top-bottom ordering

**property affine**

Underlying affine object

**property scale**

scale used in the affine transformation,  $\text{np.abs(scale)} == \text{pxsize}$

**Type** Spatial vector

**property angle**

rotation used in the affine transformation, (0 is north-up)

**Type** Angle in degree

**property pxsize**

$\text{llpixel bottom right} - \text{pixel top leftll}$  (x, y)

**Type** Spatial distance

**property pxsizex**

$\text{llpixel right} - \text{pixel leftll}$  (x)

**Type** Spatial distance

**property pxsizey**

||pixel bottom - pixel top|| (y)

**Type** Spatial distance**property pxvec**

(pixel bottom right - pixel top left)

**Type** Spatial vector**property pxtbvec**

(pixel bottom left - pixel top left)

**Type** Spatial vector**property pxlrvec**

(pixel top right - pixel top left)

**Type** Spatial vector**\_\_eq\_\_** (*other*)

Returns self.equals

**\_\_ne\_\_** (*other*)

Returns not self.equals

**share\_area** (*other*)

Binary predicate: Does other share area with self

**Parameters****other:** Footprint or shapely object**Returns**

bool

**equals** (*other*)

Binary predicate: Is other Footprint exactly equal to self

**Parameters****other:** Footprint**Returns**

bool

**almost\_equals** (*other*)

Binary predicate: Is other Footprint almost equal to self with regard to buzz.env.significant.

### Parameters

**other:** Footprint

### Returns

bool

**same\_grid** (*other*)

Binary predicate: Does other Footprint lie on the same grid as self

### Parameters

**other:** Footprint

### Returns

bool

**property shape**

(pixel per column, pixel per line)

**Type** Pixel quantities

**property meshgrid\_raster**

Compute indice matrices

### Returns

**(x, y): (np.ndarray, np.ndarray)** Raster indices matrices with shape = self.shape with dtype = env.default\_index\_dtype

**property meshgrid\_spatial**

Compute coordinate matrices

### Returns

**(x, y): (np.ndarray, np.ndarray)** Spatial coordinate matrices with shape = self.shape with dtype = float32

**meshgrid\_raster\_in** (*other*, *dtype=None*, *op=<ufunc 'floor'>*)

Compute raster coordinate matrices of *self* in *other* referential

### Parameters

**other:** Footprint

**dtype:** None or convertible to np.dtype Output dtype If None: Use buzz.env.default\_index\_dtype

**op:** None or function operating on a vector Function to apply before casting output to dtype If None: Do not transform data before casting

## Returns

**(x, y): (np.ndarray, np.ndarray)** Raster coordinate matrices with shape = self.shape with dtype = dtype

**slice\_in** (*other*, *clip=False*)

Compute location of *self* inside *other* with slice objects. If *other* and *self* do not have the same rotation, operation is undefined

## Parameters

**other: Footprint**

**clip: bool**

**False** Does nothing

**True** Clip the slices to other bounds. If *other* and *self* do not share area, at least one of the returned slice will have *slice.start == slice.stop*

## Returns

(yslice, xslice): (slice, slice)

## Example

Burn *small* into *big* if *small* is within *big* >>> big\_data[small.slice\_in(big)] = small\_data

Burn *small* into *big* where overlapping >>> big\_data[small.slice\_in(big, clip=True)] = small\_data[big.slice\_in(small, clip=True)]

**spatial\_to\_raster** (*xy*, *dtype=None*, *op=<ufunc 'floor'>*)

Convert xy spatial coordinates to raster xy indices

## Parameters

**xy: sequence of numbers of shape (... , 2)** Spatial coordinates

**dtype: None or convertible to np.dtype** Output dtype If None: Use buzz.env.default\_index\_dtype

**op: None or vectorized function** Function to apply before casting output to dtype If None: Do not transform data before casting

## Returns

**out\_xy: np.ndarray** Raster indices with shape = np.asarray(xy).shape with dtype = dtype

Prototype inspired from <https://mapbox.github.io/rasterio/api/rasterio.io.html#rasterio.io.TransformMethodsMixin.index>

**raster\_to\_spatial** (*xy*)

Convert xy raster coordinates to spatial coordinates

## Parameters

**xy:** sequence of numbers of shape  $(\dots, 2)$  Raster coordinages

## Returns

**out\_xy:** `np.ndarray` Spatial coordinates with shape = `np.asarray(xy).shape` with dtype = dtype

**find\_lines** (*arr*, *output\_offset='middle'*, *merge=True*)

Create a list of line-strings from a mask. Works with connectivity 4 and 8. The input raster is preprocessed using *skimage.morphology.thin*. The output linestrings are postprocessed using *shapely.ops.linemerge*.

**Warning:** All standalone pixels contained in *arr* will be ignored.

## Parameters

**arr:** `np.ndarray` of bool of shape (*self.shape*)

**output\_offset:** 'middle' or (*nbr*, *nbr*) Coordinate offset in meter if *middle*: substituted by *self.pxvec / 2*

## Returns

list of `shapely.geometry.LineString`

## Exemple

```
>>> import buzard as buzz
>>> import numpy as np
>>> import networkx as nx
```

```
>>> with buzz.Env(allow_complex_footprint=1):
...     a = np.asarray([
...         [0, 1, 1, 1, 0],
...         [0, 1, 0, 0, 0],
...         [0, 1, 1, 1, 0],
...         [0, 1, 0, 0, 0],
...         [0, 1, 1, 0, 0],
...     ])
...     fp = buzz.Footprint(gt=(0, 1, 0, 0, 0, 1), rsize=(a.shape))
...     lines = fp.find_lines(a, (0, 0))
...
...     # Display input / output
...     print(fp)
...     print(a.astype(int))
...     for i, l in enumerate(lines, 1):
...         print(f'edge-id:{i} of type:{type(l)} and length:{l.length}')
...         print(fp.burn_lines(l).astype(int) * i)
...
...     # Build a networkx graph
...     g = nx.Graph([(l.coords[0], l.coords[-1]) for l in lines])
```

(continues on next page)

(continued from previous page)

```

...     print(repr(g.degree))
...
Footprint(tl=(0.000000, 0.000000), scale=(1.000000, 1.000000), angle=0.000000,
↪ rsize=(5, 5))
[[0 1 1 1 0]
 [0 1 0 0 0]
 [0 1 1 1 0]
 [0 1 0 0 0]
 [0 1 1 0 0]]
edge-id:1 of type:<class 'shapely.geometry.linestring.LineString'> and_
↪ length:2.0
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 1 1 1 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
edge-id:2 of type:<class 'shapely.geometry.linestring.LineString'> and_
↪ length:3.0
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 2 0 0 0]
 [0 2 0 0 0]
 [0 2 2 0 0]]
edge-id:3 of type:<class 'shapely.geometry.linestring.LineString'> and_
↪ length:4.0
[[0 3 3 3 0]
 [0 3 0 0 0]
 [0 3 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
DegreeView({(3.0, 2.0): 1, (1.0, 2.0): 3, (2.0, 4.0): 1, (3.0, 0.0): 1})

```

**burn\_lines** (*obj*, *all\_touched=False*, *labelize=False*)  
 Creates a 2d image from lines. Uses `gdal.Polygonize`.

### Parameters

**obj:** shapely line or nested iterators over shapely lines

**labelize:** bool

- if *False*: Create a boolean mask
- if *True*: Create an integer matrix containing lines indices from order in input

### Returns

**np.ndarray**

- of bool or uint8 or int
- of shape (`self.shape`)

**find\_polygons** (*mask*)

Creates a list of polygons from a mask. Uses `gdal.Polygonize`.

**Warning:** This method is not equivalent to `cv2.findContours` that considers that pixels are points and therefore returns the indices of the pixels of the contours of the features.

This method consider that the pixels are areas and therefore returns the coordinates of the points that surrounds the features.

**Warning:** Some inputs that may produce invalid polygons (see below) are fixed with the `shapely.geometry.Polygon.buffer` method.

Shapely will issue several warnings while buzzard fixes the polygons.

```
>>> # 0 0 0 0 0 0 0
... # 0 1 1 1 0 0 0
... # 0 1 1 1 1 0 0
... # 0 1 1 1 0 1 0  <- This feature has a hole near an edge. GDAL_
↳produces a self
... # 0 1 1 1 1 1 1  touching polygon without holes. A polygon with one_
↳hole is
... # 0 1 1 1 1 1 1  returned with this method.
... # 0 0 0 0 0 0 0
```

### Parameters

**mask:** np.ndarray of bool of shape (self.shape)

### Returns

list of `shapely.geometry.Polygon`

**burn\_polygons** (*obj*, *all\_touched=False*, *labelize=False*)  
Creates a 2d image from polygons. Uses `gdal.RasterizeLayer`.

**Warning:** This method is not equivalent to `cv2.drawContours` that considers that pixels are points and therefore expect as input the indices of the outer pixels of each feature.

This method consider that the pixels are areas and therefore expect as input the coordinates of the points surrounding the features.

### Parameters

**obj:** shapely polygon or nested iterators over shapely polygons

**all\_touched:** bool Burn all polygons touched



## Returns

**np.ndarray** of bool or uint8 or int of shape (self.shape)

## Examples

```

>>> burn_polygons(poly)
>>> burn_polygons([poly, poly])
>>> burn_polygons([poly, poly, [poly, poly], multipoly, poly])

```

**tile** (*size*, *overlapx=0*, *overlapy=0*, *boundary\_effect='extend'*, *boundary\_effect\_locus='br'*)  
Tile a Footprint to a matrix of Footprint

## Parameters

**size:** (**int**, **int**) Tile width and tile height, in pixel

**overlapx:** **int** Width of a tile overlapping with each direct horizontal neighbors, in pixel

**overlapy:** **int** Height of a tile overlapping with each direct vertical neighbors, in pixel

**boundary\_effect:** {'**extend**', '**exclude**', '**overlap**', '**shrink**', '**exception**'} Behavior at boundary effect locus

- '**extend**'

- Preserve tile size
- Preserve overlapx and overlapy
- Sacrifice global bounds, results in tiles partially outside bounds at locus (if necessary)
- Preserve tile count
- Preserve boundary pixels coverage

- '**overlap**'

- Preserve tile size
- Sacrifice overlapx and overlapy, results in tiles overlapping more at locus (if necessary)
- Preserve global bounds
- Preserve tile count
- Preserve boundary pixels coverage

- '**exclude**'

- Preserve tile size
- Preserve overlapx and overlapy
- Preserve global bounds
- Sacrifice tile count, results in tiles excluded at locus (if necessary)
- Sacrifice boundary pixels coverage at locus (if necessary)

- '**shrink**'

- Sacrifice tile size, results in tiles shrunked at locus (if necessary)

- Preserve overlapx and overlap
- Preserve global bounds
- Preserve tile count
- Preserve boundary pixels coverage
- **‘exception’**
  - Raise an exception if tiles at locus do not lie inside the global bounds

**boundary\_effect\_locus:** {‘br’, ‘tr’, ‘tl’, ‘bl’} Locus of the boundary effects

- ‘br’ : Boundary effect occurs at the bottom right corner of the raster, top left coordinates are preserved
- ‘tr’ : Boundary effect occurs at the top right corner of the raster, bottom left coordinates are preserved
- ‘tl’ : Boundary effect occurs at the top left corner of the raster, bottom right coordinates are preserved
- ‘bl’ : Boundary effect occurs at the bottom left corner of the raster, top right coordinates are preserved

## Returns

### **np.ndarray**

- of dtype=object (Footprint)
- of shape (M, N)
  - with M the line count
  - with N the column count

**tile\_count** (*rowcount*, *colcount*, *overlapx=0*, *overlap=0*, *boundary\_effect='extend'*, *boundary\_effect\_locus='br'*)  
Tile a Footprint to a matrix of Footprint

## Parameters

**rowcount: int** Tile count per row

**colcount: int** Tile count per column

**overlapx: int** Width of a tile overlapping with each direct horizontal neighbors, in pixel

**overlap: int** Height of a tile overlapping with each direct vertical neighbors, in pixel

**boundary\_effect: {‘extend’, ‘exclude’, ‘overlap’, ‘shrink’, ‘exception’}** Behavior at boundary effect locus

- **‘extend’**
  - Preserve tile size
  - Preserve overlapx and overlap
  - Sacrifice global bounds, results in tiles partially outside bounds at locus (if necessary)
  - Preserve tile count

- Preserve boundary pixels coverage
- **‘overlap’**
  - Preserve tile size
  - Sacrifice overlapx and overlap<sub>y</sub>, results in tiles overlapping more at locus (if necessary)
  - Preserve global bounds
  - Preserve tile count
  - Preserve boundary pixels coverage
- **‘exclude’**
  - Preserve tile size
  - Preserve overlapx and overlap<sub>y</sub>
  - Preserve global bounds
  - Preserve tile count
  - Sacrifice boundary pixels coverage at locus (if necessary)
- **‘shrink’**
  - Sacrifice tile size, results in tiles shrunked at locus (if necessary)
  - Preserve overlapx and overlap<sub>y</sub>
  - Preserve global bounds
  - Preserve tile count
  - Preserve boundary pixels coverage
- **‘exception’**
  - Raise an exception if tiles at locus do not lie inside the global bounds

**boundary\_effect\_locus:** {‘br’, ‘tr’, ‘tl’, ‘bl’} Locus of the boundary effects

- ‘br’ : Boundary effect occurs at the bottom right corner of the raster, top left coordinates are preserved
- ‘tr’ : Boundary effect occurs at the top right corner of the raster, bottom left coordinates are preserved

**tile\_occurrence** (*size*, *pixel\_occurrence<sub>x</sub>*, *pixel\_occurrence<sub>y</sub>*, *boundary\_effect*=‘extend’, *boundary\_effect\_locus*=‘br’)

Tile a Footprint to a matrix of Footprint Each pixel occur *pixel\_occurrence<sub>x</sub>* \* *pixel\_occurrence<sub>y</sub>* times overall in the output

### Parameters

**size:** (int, int) Tile width and tile height, in pixel

**pixel\_occurrence<sub>x</sub>:** int Number of occurrence of each pixel in a line of tile

**pixel\_occurrence<sub>y</sub>:** int Number of occurrence of each pixel in a column of tile

**boundary\_effect:** {‘extend’, ‘exclude’, ‘overlap’, ‘shrink’, ‘exception’} Behavior at boundary effect locus

- ‘extend’

- Preserve tile size
- Preserve overlapx and overlapy
- Sacrifice global bounds, results in tiles partially outside bounds at locus (if necessary)
- Preserve tile count
- Preserve boundary pixels coverage
- **‘overlap’**
  - Preserve tile size
  - Sacrifice overlapx and overlapy results in tiles overlapping more at locus (if necessary)
  - Preserve global bounds
  - Preserve tile count
  - Preserve boundary pixels coverage
- **‘exclude’**
  - Preserve tile size
  - Preserve overlapx and overlapy
  - Preserve global bounds
  - Sacrifice tile count, results in tiles excluded at locus (if necessary)
  - Sacrifice boundary pixels coverage at locus (if necessary)
- **‘shrink’**
  - Sacrifice tile size, results in tiles shrunked at locus (if necessary)
  - Preserve overlapx and overlapy
  - Preserve global bounds
  - Preserve tile count
  - Preserve boundary pixels coverage
- **‘exception’** Raise an exception if tiles at locus do not lie inside the global bounds

**boundary\_effect\_locus: {‘br’, ‘tr’, ‘tl’, ‘bl’}** Locus of the boundary effects

- ‘br’ : Boundary effect occurs at the bottom right corner of the raster top left coordinates are preserved
- ‘tr’ : Boundary effect occurs at the top right corner of the raster, bottom left coordinates are preserved
- ‘tl’ : Boundary effect occurs at the top left corner of the raster, bottom right coordinates are preserved
- ‘bl’ : Boundary effect occurs at the bottom left corner of the raster, top right coordinates are preserved

## Returns

### np.ndarray

- of dtype=object (Footpr
- int) - of shape (M, N)
- with M the line count
- with N the column count

### \_\_str\_\_()

Return str(self).

### \_\_repr\_\_()

Return repr(self).

### \_\_reduce\_\_()

Helper for pickle.

### \_\_hash\_\_()

Return hash(self).

### forward\_conv2d(kernel\_size, stride=1, padding=0, dilation=1)

Shift, scale and dilate the Footprint as if it went through a 2d convolution kernel.

The arithmetic followed is the one from *pytorch*, but other deep-learning libraries mostly follow the same arithmetic.

This function is a *many to one* mapping, two footprints with different *rsizes* can produce the same Footprint when *stride > 1*.

## Parameters

**kernel\_size:** int or (int, int) See *torch.nn.Conv2d* documentation.

**stride:** int or (int, int) See *torch.nn.Conv2d* documentation.

**padding:** int or (int, int) See *torch.nn.Conv2d* documentation.

**dilation:** int or (int, int) See *torch.nn.Conv2d* documentation.

## Returns

Footprint

## Example

```
>>> fp0 = buzz.Footprint(tl=(0, 0), size=(1024, 1024), rsize=(1024, 1024))
... fp1 = fp0.forward_conv2d(kernel_size=2, stride=2)
... print(fp1)
Footprint(tl=(0.5, -0.5), size=(1024, 1024), rsize=(512, 512))
```

### backward\_conv2d(kernel\_size, stride=1, padding=0, dilation=1)

Shift, scale and dilate the Footprint as if it went backward through a 2d convolution kernel.

The arithmetic followed is the one from *pytorch*, but other deep-learning libraries mostly follow the same arithmetic.

This function is a *one to one* mapping, two different input footprints will produce two different output Footprints. It means that the *backward\_conv2d* of a *forward\_conv2d* may not reproduce the initial Footprint, some pixels on the bottom and right edges may be missing.

### Parameters

**kernel\_size:** int or (int, int) See *torch.nn.Conv2d* documentation.

**stride:** int or (int, int) See *torch.nn.Conv2d* documentation.

**padding:** int or (int, int) See *torch.nn.Conv2d* documentation.

**dilation:** int or (int, int) See *torch.nn.Conv2d* documentation.

### Returns

Footprint

### Example

```
>>> fp1 = buzz.Footprint(tl=(0.5, -0.5), size=(1024, 1024), rsize=(512, 512))
... fp0 = fp1.backward_conv2d(kernel_size=2, stride=2)
... print(fp0)
Footprint(tl=(0, 0), size=(1024, 1024), rsize=(1024, 1024))
```

**forward\_convtranspose2d** (*kernel\_size, stride=1, padding=0, dilation=1, output\_padding=0*)

Shift, scale and dilate the Footprint as if it went through a 2d transposed convolution kernel.

The arithmetic followed is the one from *pytorch*, but other deep-learning libraries mostly follow the same arithmetic.

A 2d transposed convolution has 4 internal steps: 1. Apply stride (i.e. interleave the input pixels with zeroes) 2. Add padding 3. Apply a 2d convolution with stride=1 and pad=0 4. Add output-padding

This function is a *one to one* mapping, two different input footprints will produce two different output Footprints.

### Parameters

**kernel\_size:** int or (int, int) See *torch.nn.ConvTranspose2d* documentation.

**stride:** int or (int, int) See *torch.nn.ConvTranspose2d* documentation.

**padding:** int or (int, int) See *torch.nn.ConvTranspose2d* documentation.

**dilation:** int or (int, int) See *torch.nn.ConvTranspose2d* documentation.

**output\_padding:** int or (int, int) See *torch.nn.ConvTranspose2d* documentation.

## Returns

Footprint

## Example

```
>>> fp0 = buzz.Footprint(tl=(0, 0), size=(1024, 1024), rsize=(512, 512))
... fp1 = fp0.forward_convtranspose2d(kernel_size=3, stride=2, padding=1)
... print(fp1)
Footprint(tl=(0, 0), size=(1023, 1023), rsize=(1023, 1023))
```

**backward\_convtranspose2d** (*kernel\_size, stride=1, padding=0, dilation=1, output\_padding=0*)

Shift, scale and dilate the Footprint as if it went backward through a 2d transposed convolution kernel.

The arithmetic followed is the one from *pytorch*, but other deep-learning libraries mostly follow the same arithmetic.

A 2d transposed convolution has 4 internal steps: 1. Apply stride (interleave the input pixels with zeroes) 2. Add padding 3. Apply a 2d convolution stride:1, pad:0 4. Add output-padding

This function is a *one to one* mapping, two different input Footprints will produce two different output Footprints.

## Parameters

**kernel\_size: int or (int, int)** See *torch.nn.ConvTranspose2d* documentation.

**stride: int or (int, int)** See *torch.nn.ConvTranspose2d* documentation.

**padding: int or (int, int)** See *torch.nn.ConvTranspose2d* documentation.

**dilation: int or (int, int)** See *torch.nn.ConvTranspose2d* documentation.

**output\_padding: int or (int, int)** See *torch.nn.ConvTranspose2d* documentation.

## Returns

Footprint

## Example

```
>>> fp0 = buzz.Footprint(tl=(0, 0), size=(1023, 1023), rsize=(1023, 1023))
... fp1 = fp0.backward_convtranspose2d(kernel_size=3, stride=2, padding=1)
... print(fp1)
Footprint(tl=(0, 0), size=(1024, 1024), rsize=(512, 512))
```

## 1.4 Env

**class** `buzzard.Env` (\*\*kwargs)

Context manager to update buzzard's states. Can also be used as a decorator.

### Parameters

**significant: int** Number of significant digits for floating point comparisons Initialized to *9.0* see: <https://github.com/earthcube-lab/buzzard/wiki/Precision-system> see: <https://github.com/earthcube-lab/buzzard/wiki/Floating-Point-Considerations>

**default\_index\_dtype: convertible to np.dtype** Default numpy return dtype for array indices. Initialized to *np.int32* (signed to allow negative indices by default)

**allow\_complex\_footprint: bool** Whether to allow non north-up / west-left Footprints Initialized to *False*

### Examples

```
>>> import buzzard as buzz
>>> with buzz.Env(default_index_dtype='uint64'):
...     ds = buzz.Dataset()
...     dsm = ds.aopen_raster('dsm', 'path/to/dsm.tif')
...     x, y = dsm.meshgrid_raster
...     print(x.dtype)
numpy.uint64
```

```
>>> @buzz.Env(allow_complex_footprint=True)
... def main():
...     fp = buzz.Footprint(rsize=(10, 10), gt=(100, 1, 0, 100, 0, 1))
```

`__enter__` ()

`__exit__` (exc\_type=None, exc\_val=None, exc\_tb=None)

`__call__` (fn)

Call self as a function.

`buzzard.env` = <buzzard.\_env.\_CurrentEnv object>

Namespace to access current values of buzzard's environment variable (see `buzz.Env`)

### Example

```
>>> buzz.env.significant
8.0
```



## 1.5 Misc.

`buzzard.open_raster(*args, **kwargs)`

Shortcut for `Dataset().aopen_raster`

```
>>> help(Dataset.open_raster)
```

### See Also

- `Dataset.open_raster()`
- `Dataset.aopen_raster()`

`buzzard.create_raster(*args, **kwargs)`

Shortcut for `Dataset().acreate_raster`

```
>>> help(Dataset.create_raster)
```

### See Also

- `Dataset.create_raster()`
- `Dataset.acreate_raster()`

`buzzard.wrap_numpy_raster(*args, **kwargs)`

Shortcut for `Dataset().awrap_numpy_raster`

```
>>> help(Dataset.wrap_numpy_raster)
```

### See Also

- `Dataset.wrap_numpy_raster()`
- `Dataset.awrap_numpy_raster()`

`buzzard.open_vector(*args, **kwargs)`

Shortcut for `Dataset().aopen_vector`

```
>>> help(Dataset.open_vector)
```

### See Also

- `Dataset.open_vector()`
- `Dataset.aopen_vector()`

`buzzard.create_vector(*args, **kwargs)`

Shortcut for `Dataset().acreate_vector`

```
>>> help(Dataset.create_vector)
```

### See Also

- `Dataset.create_vector()`
- `Dataset.create_vector()`

`buzzard.utils.concat_arrays(fp, array_per_fp, ...)`

Concatenate arrays from `array_per_fp` to form `fp`.

This function is meant to be fed to the `merge_arrays` parameter when constructing a recipe.

## CAVEATS, FAQs AND DESIGN CHOICES

Buzzard has a lot of ambition but is still a young library with several caveats. Are you currently trying to determine if buzzard is the right choice for your project? We got you covered and listed here the use-cases that are currently poorly supported. The rest is a bliss!

### 2.1 Caveat List

#### 2.1.1 Installation

→ `buzzard` installation is complex because of the `GDAL` and `rtree` dependencies.

→ The `anaconda` package does not exist

#### 2.1.2 Rasters

→ Reading a raster file is currently internally performed by calls to `GDAL` drivers, and it might be too slow under certain circumstances. Tweaking the `GDAL_CACHEMAX` variable may improve performances.

→ On-the-fly reprojections is an ambitious feature of `buzzard`, but this feature only reaches its full potential with vectorial data. On-the-fly raster reprojections are currently partially supported. Those only work if the reprojection preserve angles, if not an exception is raised.

#### 2.1.3 Floating point precision losses

→ The biggest plague of a GIS library is the floating point precision losses. On one hand those losses cannot be avoided (such as in a reprojection operation), and on the other hand certain operations can only be performed with noise-free numbers (such as the floor or ceil operations). The only solution is to round those numbers before critical operations. `buzzard` has its own way of dealing with this problem: it introduces a global variable to define the number of significant digits that should be considered as noise-less (9 by default).

This way `buzzard` tries to catch the errors early and raise exceptions. But despite all those efforts some bugs still occur when the noise reaches the significant digits, resulting in strange exceptions being raise.

However those bugs only occur when manipulating very small pixels along with very large coordinates, which is not usual (the ratio `coordinate/pixel-size` should not exceede `10 ** env.significant`).

### 2.1.4 The Footprint class

→ The Footprint class is long to instantiate (~0.5ms), several use cases involving masses of Footprints are impractical because of this.

→ The Footprint class is the key feature of buzzard, but its specifications are broader than its unit tests: the non-north-up rasters are not fully unit tested. To instantiate such a Footprint the `buzz.env.allow_complex_footprint` should be set to `True`. However those Footprints should work fine in general

→ The Footprint class lacks some higher level constructors to make several common construction schemes easier. However by using the intersection method of a Footprint on itself and tweaking the 3 optional parameters covers most of the missing use-cases.

### 2.1.5 The async rasters

→ Most of the async rasters as advertised in the doc or the examples are not yet implemented. Only the cached raster recipes are.

→ Using cached raster recipes has a side effect on a file system. Using a single cache directory from two different programs at the same time is an undefined behavior. Although it works fine when the cache files are already instantiated.

→ The scheduler that was written to support the async rasters is not proven to be bug free. Although it is filled with assertions that will most likely catch any remaining bug.

## 2.2 FAQs and design choices

The following list contains the FAQs or features that are often mistaken as bugs ;)

→ Why buzzard instead of fiona or rasterio that are much more mature and straightforward libraries?

The answer is simple: as soon as you are working with large images, or with geometries alongside images, you can benefit from the higher level abstractions that buzzard provides.

→ Why can't I simply reproject shapely geometries using buzzard?

Because buzzard does not aim to replace pyproj. When using the classic stack, each of osgeo's lib has its own wrapper:

- GEOS -> shapely
- OGR -> fiona
- GDAL -> rio
- OSR -> pyproj

Buzzard is transversal, it wraps enough OGR, GDAL and OSR so that you don't have to import those most of the time. Some known exceptions are:

- Raster reprojection that does not preserve angles
- Shapely objects reprojection
- Contour lines generation

It might be the case that someday buzzard provides a transversal feature that replaces pyproj but nothing is planned.

→ In buzzard, all sources (such as raster and vector files) are tied to a Dataset object. This is a design choice that has several advantages now and even more advantages in the long term. See the Dataset's docstring.

→ buzzard is a binding for GDAL, but all the features that allows editing the attributes of an opened file are not exposed in buzzard. The wish here is to make buzzard as [functional](#) as possible.

→ The `with Dataset.close as ds:` syntax is chosen over the `with Dataset as ds:` syntax in order to stay consistent with the `with Source.close as src:` syntax, that itself exist because of the need for disambiguation with this other feature: `with Source.delete as src:.`

→ The Footprint class is an [immutable](#) object. This is not a bug.

→ Why is the Footprint class not directly implementing a shapely Polygon?

In the early versions of buzzard, it was the case. But method name conflicts became a big problem. And overall, it was not that useful. You can still use `Footprint.poly` to convert a Footprint to a shapely Polygon.

→ Why support non-north-up Footprints?

It was harder to design but cleaner in the end. Now that it is (mostly - missing unit tests at the moment) supported there is a hope that it creates new use cases.

→ Why are the `get_data` and `set_data` methods of a raster so complex?

Those methods accept **any** Footprint as a parameter, it includes Footprints that don't share alignment/scale/rotation/bounds with the raster source. It allows the user to forget about the file when designing a piece of code. The downside of this feature is that the user is not aware when a resource consuming resampling is performed. To avoid this problem, the Dataset class is by default configured to raise an error when an interpolation occurs.

→ If you ever wander in the buzzard source code you may notice that the Dataset class holds pointers to Source objects and vice versa (through dependency injection). This recursive dependency reveal the design choice of making the Dataset and the Source classes a **single** class. The Source objects should be seen as extensions of a Dataset object.

→ If you ever wander in the buzzard source code you will notice a complex separation of concern scheme in which a class is split between a *facade* and a *back* class.

This separation exists in order to allow garbage collection to be made, even if the Dataset instantiates a scheduler on a separate thread. The facade classes are manipulated by the user and have pointers towards the back classes, and the later have no references to the facade, while the scheduler only have pointers to the *back* classes. This way, when the facade are collected, the back are collected too. This separation also allows us to perform parameter checking only once in the facade classes, and then call the appropriate back implementation using [dynamic dispatch](#).



**INDICES AND TABLES:**

- genindex
- modindex
- search





## Symbols

`__contains__()` (*buzzard.Dataset* method), 8  
`__contains__()` (*buzzard.PoolsContainer* method), 10  
`__del__()` (*buzzard.Dataset* method), 8  
`__getitem__()` (*buzzard.Dataset* method), 8  
`__getitem__()` (*buzzard.PoolsContainer* method), 10  
`__iter__()` (*buzzard.PoolsContainer* method), 10  
`__len__()` (*buzzard.Dataset* method), 9  
`__len__()` (*buzzard.PoolsContainer* method), 10

## A

`acreate_cached_raster_recipe()` (*buzzard.Dataset* method), 20  
`acreate_raster()` (*buzzard.Dataset* method), 13  
`acreate_vector()` (*buzzard.Dataset* method), 23  
`activate_all()` (*buzzard.Dataset* method), 9  
`active_count()` (*buzzard.Dataset* property), 9  
`alias()` (*buzzard.PoolsContainer* method), 9  
`aopen_raster()` (*buzzard.Dataset* method), 12  
`aopen_vector()` (*buzzard.Dataset* method), 21  
`awrap_numpy_raster()` (*buzzard.Dataset* method), 14

## C

`close()` (*buzzard.Dataset* property), 8  
`concat_arrays()` (*in module buzzard.utils*), 86  
`create_cached_raster_recipe()` (*buzzard.Dataset* method), 18  
`create_raster()` (*buzzard.Dataset* method), 11  
`create_raster()` (*in module buzzard*), 85  
`create_raster_recipe()` (*buzzard.Dataset* method), 14  
`create_vector()` (*buzzard.Dataset* method), 21  
`create_vector()` (*in module buzzard*), 85

## D

`Dataset` (*class in buzzard*), 3  
`deactivate_all()` (*buzzard.Dataset* method), 9

## E

`env` (*in module buzzard*), 84

## I

`items()` (*buzzard.Dataset* method), 9

## K

`keys()` (*buzzard.Dataset* method), 9

## M

`manage()` (*buzzard.PoolsContainer* method), 9

## O

`open_raster()` (*buzzard.Dataset* method), 10  
`open_raster()` (*in module buzzard*), 85  
`open_vector()` (*buzzard.Dataset* method), 20  
`open_vector()` (*in module buzzard*), 85

## P

`pools()` (*buzzard.Dataset* property), 9  
`PoolsContainer` (*class in buzzard*), 9  
`proj4()` (*buzzard.Dataset* property), 9

## V

`values()` (*buzzard.Dataset* method), 9

## W

`wkt()` (*buzzard.Dataset* property), 9  
`wrap_numpy_raster()` (*buzzard.Dataset* method), 13  
`wrap_numpy_raster()` (*in module buzzard*), 85