GeoPySpark Documentation

Release 0.4.2

Jacob Bouffard, James McClean, Eugene Cheipesh

Jun 07, 2018

Home

GeoPySpark is a Python language binding library of the Scala library, GeoTrellis. Like GeoTrellis, this project is released under the Apache 2 License.

GeoPySpark seeks to utilize GeoTrellis to allow for the reading, writing, and operating on raster data. Thus, its able to scale to the data and still be able to perform well.

In addition to raster processing, GeoPySpark allows for rasters to be rendered into PNGs. One of the goals of this project to be able to process rasters at web speeds and to perform batch processing of large data sets.

CHAPTER 1

Why GeoPySpark?

Raster processing in Python has come a long way; however, issues still arise as the size of the dataset increases. Whether it is performance or ease of use, these sorts of problems will become more common as larger amounts of data are made available to the public.

One could turn to GeoTrellis to resolve the aforementioned problems (and one should try it out!), yet this brings about new challenges. Scala, while a powerful language, has something of a steep learning curve. This can put off those who do not have the time and/or interest in learning a new language.

By having the speed and scalability of Scala and the ease of Python, GeoPySpark is then the remedy to this predicament.

CHAPTER 2

Contact and Support

If you need help, have questions, or would like to talk to the developers (let us know what you're working on!) you can contact us at:

- Gitter
- Mailing list

As you may have noticed from the above links, those are links to the GeoTrellis Gitter channel and mailing list. This is because this project is currently an offshoot of GeoTrellis, and we will be using their mailing list and gitter channel as a means of contact. However, we will form our own if there is a need for it.

2.1 Changelog

2.1.1 0.4.2

Experimental New Features

Creating a RasterLayer From URIs Using rasterio

While the ability to create a `RasterLayer` from `URIs` already exists with the `geopyspark.geotrellis.geotiff.get` function, it is limited to just working with GeoTiffs. However, with the new `geopyspark.geotrellis.rasterio` module, it is now possible to create `RasterLayers` from different file types.

```
uris = ["file://images/image_1.jp2", "file://images/image_2.jp2"]

raster_layer = gps.rasterio.get(uris)
```

Note: This feature is experimental, and will most likely be improved and/or changed in the future releases of GeoPySpark.

2.1.2 0.4.1

Bug Fixes

There was a bug in the Scala backend in 0.4.0 that caused certain layers on S3 to not be read. This has since been resolved and 0.4.1 will have this fixed Scala backend. No other notable changes/fixes have been done between 0.4.0 and 0.4.1.

2.1.3 0.4.0

New Features

Rasterizing an RDD[Geometry]

Users can now rasterize an `RDD[shapely.geometry]` via the `rasterize` method.

```
# A Python RDD that contains shapely geomtries
geometry_rdd = ...

gps.rasterize(geoms=geometry_rdd, crs="EPSG:3857", zoom=11, fill_value=1)
```

ZFactor Calculator

`zfactor_lat_lng_caculator` and `zfactor_caclulator` are two new functions that will caculate the the the `zfactor` for each `Tile` in a layer during the `slope` or `hillshade` operations. This is better than using a single `zfactor` for all `Tiles` as `Tiles` at different lattitdues require different `zfactors`.

As mentioned above, there are two different forms of the calculator: `zfactor_lat_lng_calculator` and `zfactor_calculator`. The former being used for layers that are in the LatLng projection while the latter for layers in all other projections.

```
# Using the zfactor_lat_lng_calculator

# Create a zfactor_lat_lng_calculator which uses METERS for its calcualtions
calculator = gps.zfactor_lat_lng_calculator(gps.METERS)

# A TiledRasterLayer which contains elevation data
tiled_layer = ...

# Calcualte slope of the layer using the calcualtor
tiled_layer.slope(calculator)

# Using the zfactor_calculator

# We must provide a dict that maps lattitude to zfactor for our
# given projection. Linear interpolation will be used on these
# values to produce the correct zfactor for each Tile in the
# layer.

mapped_factors = {
  0.0: 0.1,
  10.0: 1.5,
  15.0: 2.0,
```

(continues on next page)

(continued from previous page)

```
  20.0, 2.5
}

# Create a zfactor_calculator using the given mapped factors
calculator = gps.zfactor_calculator(mapped_factors)
```

PartitionStragies

With this release of GeoPySpark comes three different parition strategies: `HashPartitionStrategy`, `SpatialPartitionStrategy`, and `SpaceTimePartitionStrategy`. All three of these are used to partition a layer given their specified inputs.

HashPartitionStrategy

`HashPartitionStrategy` is a partition strategy that uses Spark's `HashPartitioner` to partition a layer. This can be used on either `SPATIAL` or `SPACETIME` layers.

```
# Creates a HashPartitionStrategy with 128 partitions
gps.HashPartitionStrategy(num_partitions=128)
```

SpatialPartitionStrategy

`SpatialPartitionStrategy` uses GeoPySpark's `SpatialPartitioner` during partitioning of the layer. This strategy will try and partition the `Tiles` of a layer so that those which are near each other spatially will be in the same partition. This will only work on `SPATIAL` layers.

```
# Creates a SpatialPartitionStrategy with 128 partitions
gps.SpatialPartitionStrategy(num_partitions=128)
```

SpaceTimePartitionStrategy

`SpaceTimePartitionStrategy` uses GeoPySpark's `SpaceTimePartitioner` during partitioning of the layer. This strategy will try and partition the `Tiles` of a layer so that those which are near each other spatially and temporally will be in the same partition. This will only work on `SPACETIME` layers.

```
# Creates a SpaceTimePartitionStrategy with 128 partitions
# and temporal resolution of 5 weeks. This means that
# it will try and group the data in units of 5 weeks.
gps.SpaceTimePartitionStrategy(time_unit=gps.WEEKS, num_partitions=128, time_
↪resolution=5)
```

Other New Features

- tobler method for TiledRasterLayer
- slope method for TiledRasterLayer
- local_max method for TiledRasterLayer

- mask layers by RDD[Geometry]
- with_no_data method for RasterLayer and TiledRasterLayer
- `partitionBy` method for `RasterLayer` and `TiledRasterLayer`
- `get_partition_strategy` method for `CachableLayer`

Bug Fixes

- TiledRasterLayer reproject bug fix
- TMS display fix
- CellType representation and conversion fixes
- get_point_values will now return the correct number of results for temporal layers
- Reading layers and values from Accumulo fix
- time_intervals will now enumerate correctly in catalog.query
- TileReader will now read the correct attribures file

2.1.4 0.3.0

New Features

Aggregating a Layer By Cell

It is now possible to aggregate the cells of all values that share a key in a layer via the `aggregate_by_cell` method. This method is useful when you have a layer where you want to reduce all of the values by their key.

```
# A tiled layer which contains duplicate keys with different values
# that we'd like to reduce so that there is one value per key.
tiled_layer = ...

# This will compute the aggregate SUM of each cell of values that share
# a key within the layer.
tiled_layer.aggregate_by_cell(gps.Operation.SUM)

# Similar to the above command, only this one is finding the STANDARD_DEVIATION
# for each cell.
tiled_layer.aggregate_by_cell(gps.Operation.STANDARD_DEVIATION)
```

Unioning Layers Together

Through the `union` method, it is now possible to union together an arbitrary number of either `RasterLayers` or `TiledRasterLayers`.

```
# Layers to be unioned together
layers = [raster_layer_1, raster_layer_2, raster_layer_3]

unioned_layers = gps.union(layers)
```

Getting Point Values From a Layer

By using the `get_point_values` method, one can retrieve data points that falls on or near a given point.

```
from shapely.geometry import Point

# The points we'd like to collect data at
p1 = Point(0, 0)
p2 = Point(1, 1)
p3 = Point(10, 10)

# The tiled layer which will be queried
tiled_layer = ...

tiled_layer.get_point_values([p1, p2, p3])
```

The above code will return a `[(Point, [float])]` where each point given will be paired with all of the values it covers (one for each band of the Tile).

It is also possible to pass in a `dict` to `get_point_values`.

```
labeled_points = {'p1': p1, 'p2': p2, 'p3': p3}

tiled_layer.get_point_values(labeled_points)
```

This will return a `{k: (Point, [float])}` which is similar to the above code only now the `(Point, [float])` is the value of the key that point had in the input `dict`.

Combining Bands of Multiple Layers

`combine_bands` will concatenate the bands of values that share a key together to produce a new, single value. This new Tile will contain all of the bands from all of the values that shared a key from the given layers.

This method is most useful when you have multiple layers that contain a single band from a multiband image; and you'd like to combine them together so that all or some of the bands are available from a single layer.

```
# Three different layers that contain a single band from the
# same scene
band_1_layer = ...
band_2_layer = ...
band_3_layer = ...

# combined_layer will have values that contain three bands: the first
# from band_1_layer, the second from band_2_layer, and the last from
# band_3_layer
combined_layer = gps.combine_bands([band_1_layer, band_2_layer, band_3_layer])
```

Other New Features

- Merge method for RasterLayer and TiledRasterLayer
- Filter a RasterLayer or a TiledRasterLayer by time
- Polygonal Summary on all bands
- Better temporal resolution control when writing layers

- TiledRasterLayers can now perform the abs local operation
- TiledRasterLayers can now perform the ** local operation

Bug Fixes

- LayerType creation issue
- tuple serializer creation fix
- The TMS can now read from MultibandTile catalogs
- tileToLayout bug
- additional_jar_dirs fix
- stitch and saveStitch now work with MultibandTiles

2.1.5 0.2.2

0.2.2 fixes the naming issue brought about in 0.2.1 where the backend jar and the docs had the incorrect version number.

geopyspark

- Fixed version numbers for docs and jar.

2.1.6 0.2.1

0.2.1 adds two major bug fixes for the `catalog.query` and `geotiff.get` functions as well as a few other minor changes/additions.

geopyspark

- Updated description in `setup.py`.

geopyspark.geotrellis

- Fixed a bug in `catalog.query` where the query would fail if the geometry used for querying was in a different projection than the source layer.
- `partition_bytes` can now be set in the `geotiff.get` function when reading from S3.
- Setting `max_tile_size` and `num_partitions` in `geotiff.get` will now work when trying to read geotiffs from S3.

2.1.7 0.2.0

The second release of GeoPySpark has brought about massive changes to the library. Many more features have been added, and some have been taken away. The API has also been overhauld, and code written using the 0.1.0 code will not work with this version.

Because so much has changed over these past few months, only the most major changes will be discussed below.

geopyspark

- Removed `GeoPyContext`.
- Added `geopyspark_conf` function which is used to create a `SparkConf` for GeoPySpark.

- Changed how the environemnt is constructed when using GeoPySpark.

geopyspark.geotrellis

- A `SparkContext` instance is no longer needs to be passed in for any class or function.
- Renamed `RasterRDD` and `TiledRasterRDD` to `RasterLayer` and `TiledRasterLayer`.
- Changed how `tile_to_layout` and `reproject` work.
- Broked out `rasterize`, `hillshade`, `cost_distance`, and `euclidean_distance` into their own, respective modules.
- Added the `Pyramid` class to `layer.py`.
- Renamed `geotiff_rdd` to `geotiff`.
- Broke out the options in `geotiff.get`.
- Constants are now orginized by enum classes.
- Avro is no longer used for serialization/deserialization.
- ProtoBuf is now used for serialization/deserialization.
- Added the `render` module.
- Added the `color` mdoule.
- Added the `histogram` moudle.

Documentation

- Updated all of the docstrings to reflect the new changes.
- All of the documentation has been updated to reflect the new chnagtes.
- Example jupyter notebooks have been added.

2.1.8 0.1.0

The first release of GeoPySpark! After being in development for the past 6 months, it is now ready for its initial release! Since nothing has been changed or updated per se, we'll just go over the features that will be present in 0.1.0.

geopyspark.geotrellis

- Create a `RasterRDD` from GeoTiffs that are stored locally, on S3, or on HDFS.
- Serialize Python RDDs to Scala and back.
- Perform various tiling operations such as `tile_to_layout`, `cut_tiles`, and `pyramid`.
- Stitch together a `TiledRasterRDD` to create one `Raster`.
- `rasterize` geometries and turn them into `RasterRDD`.
- `reclassify` values of Rasters in RDDs.
- Calculate `cost_distance` on a `TiledRasterRDD`.
- Perform local and focal operations on `TiledRasterRDD`.
- Read, write, and query GeoTrellis tile layers.
- Read tiles from a layer.
- Added `PngRDD` to make rendering to PNGs more efficient.
- Added `RDDWrapper` to provide more functionality to the RDD classes.

- Polygonal summary methods are now available to `TiledRasterRDD`.
- Euclidean distance added to `TiledRasterRDD`.
- Neighborhoods submodule added to make focal operations easier.

geopyspark.command

- GeoPySpark can now use a script to download the jar. Used when installing GeoPySpark from pip.

Documentation

- Added docstrings to all python classes, methods, etc.
- Core-Concepts, rdd, geopycontext, and catalog.
- Ingesting and creating a tile server with a greyscale raster dataset.
- Ingesting and creating a tile server with data from Sentinel.

2.2 Contributing

We value all kinds of contributions from the community, not just actual code. Perhaps the easiest and yet one of the most valuable ways of helping us improve GeoPySpark is to ask questions, voice concerns or propose improvements on the GeoTrellis Mailing List. As of now, we will be using this to interact with our users. However, this could change depending on the volume/interest of users.

If you do like to contribute actual code in the form of bug fixes, new features or other patches this page gives you more info on how to do it.

2.2.1 Building GeoPySpark

Ensure you have the **‘project dependencies<https://github.com/locationtech-labs/geopyspark/blob/master/README.rst#requirements>‘_** installed on your machine.

Then follow the **‘Installing for Developers<https://github.com/locationtech-labs/geopyspark/blob/master/README.rst#installing-for-developers>‘_** instructions in the project README.

2.2.2 Style Guide

We try to follow the PEP 8 Style Guide for Python Code as closely as possible, although you will see some variations throughout the codebase. When in doubt, follow that guide.

2.2.3 Git Branching Model

The GeoPySpark team follows the standard practice of using the `master` branch as main integration branch.

2.2.4 Git Commit Messages

We follow the ‘imperative present tense’ style for commit messages. (e.g. “Add new EnterpriseWidgetLoader instance”)

2.2.5 Issue Tracking

If you find a bug and would like to report it please go there and create an issue. As always, if you need some help join us on Gitter to chat with a developer. As with the mailing list, we will be using the GeoTrellis Gitter channel until the need arises to form our own.

2.2.6 Pull Requests

If you'd like to submit a code contribution please fork GeoPySpark and send us pull request against the `master` branch. Like any other open source project, we might ask you to go through some iterations of discussion and refinement before merging.

As part of the Eclipse IP Due Diligence process, you'll need to do some extra work to contribute. This is part of the requirement for Eclipse Foundation projects (see this page in the Eclipse wiki You'll need to sign up for an Eclipse account **with the same email you commit to github with**. See the `Eclipse Contributor Agreement` text below. Also, you'll need to signoff on your commits, using the `git commit -s` flag. See https://help.github.com/articles/signing-tags-using-gpg/ for more info.

2.2.7 Eclipse Contributor Agreement (ECA)

Contributions to the project, no matter what kind, are always very welcome. Everyone who contributes code to GeoTrellis will be asked to sign the Eclipse Contributor Agreement. You can electronically sign the Eclipse Contributor Agreement here.

2.2.8 Editing these Docs

Contributions to these docs are welcome as well. To build them on your own machine, ensure that `sphinx` and `make` are installed.

Installing Dependencies

Ubuntu 16.04

```
> sudo apt-get install python-sphinx python-sphinx-rtd-theme
```

Arch Linux

```
> sudo pacman -S python-sphinx python-sphinx_rtd_theme
```

MacOS

`brew` doesn't supply the sphinx binaries, so use `pip` here.

Pip

```
> pip install sphinx sphinx_rtd_theme
```

Building the Docs

Assuming you've cloned the GeoTrellis repo, you can now build the docs yourself. Steps:

1. Navigate to the `docs/` directory
2. Run `make html`
3. View the docs in your browser by opening `_build/html/index.html`

Note: Changes you make will not be automatically applied; you will have to rebuild the docs yourself. Luckily the docs build in about a second.

File Structure

There is currently not a file structure in place for docs. Though, this will change soon.

2.3 Core Concepts

Because GeoPySpark is a binding of an existing project, GeoTrellis, some terminology and data representations have carried over. This section seeks to explain this jargon in addition to describing how GeoTrellis types are represented in GeoPySpark.

Before begining, all examples in this guide need the following boilerplate code:

```
import datetime
import numpy as np
import geopyspark as gps
```

2.3.1 Rasters

GeoPySpark differs in how it represents rasters from other geo-spatial Python libraries like rasterIO. In GeoPySpark, they are represented by the *`Tile`* class. This class contains a numpy array (refered to as `cells`) that represents the cells of the raster in addition to other information regarding the data. Along with `cells`, `Tile` can also have the `no_data_value` of the raster.

Note: All rasters in GeoPySpark are represented as having multiple bands, even if the original raster just contained one.

```
arr = np.array([[[0, 0, 0, 0],
                 [1, 1, 1, 1],
                 [2, 2, 2, 2]]], dtype=np.int16)

# The resulting Tile will set -10 as the no_data_value for the raster
gps.Tile.from_numpy_array(numpy_array=arr, no_data_value=-10)
```

```
# The resulting Tile will have no no_data_value
gps.Tile.from_numpy_array(numpy_array=arr)
```

2.3.2 Extent

Describes the area on Earth a raster represents. This area is represented by coordinates that are in some Coordinate Reference System. Thus, depending on the system in use, the values that outline the *Extent* can vary. `Extent` can also be refered to as a *bounding box*.

Note: The values within the `Extent` must be `float`s and not `double`s.

```
extent = gps.Extent(0.0, 0.0, 10.0, 10.0)
extent
```

2.3.3 ProjectedExtent

ProjectedExtent describes both the area on Earth a raster represents in addition to its CRS. Either the EPSG code or a proj4 string can be used to indicate the CRS of the `ProjectedExtent`.

```
# Using an EPSG code

gps.ProjectedExtent(extent=extent, epsg=3857)
```

```
# Using a Proj4 String

proj4 = "+proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +a=6378137 +b=6378137 +towgs84=0,0,0,
↪0,0,0,0 +units=m +no_defs "
gps.ProjectedExtent(extent=extent, proj4=proj4)
```

2.3.4 TemporalProjectedExtent

Similar to `ProjectedExtent`, *TemporalProjectedExtent* describes the area on Earth the raster represents, its CRS, and the time the data was represents. This point of time, called `instant`, is an instance of `datetime.datetime`.

```
time = datetime.datetime.now()
gps.TemporalProjectedExtent(extent=extent, instant=time, epsg=3857)
```

2.3.5 TileLayout

TileLayout describes the grid which represents how rasters are orginized and assorted in a layer. `layoutCols` and `layoutRows` detail how many columns and rows the grid itself has, respectively. While `tileCols` and `tileRows` tell how many columns and rows each individual raster has.

```
# Describes a layer where there are four rasters in a 2x2 grid. Each raster has 256␣
↪cols and rows.

tile_layout = gps.TileLayout(layoutCols=2, layoutRows=2, tileCols=256, tileRows=256)
tile_layout
```

2.3.6 LayoutDefinition

LayoutDefinition describes both how the rasters are orginized in a layer as well as the area covered by the grid.

```
layout_definition = gps.LayoutDefinition(extent=extent, tileLayout=tile_layout)
layout_definition
```

2.3.7 Tiling Strategies

It is often the case that the exact layout of the layer is unknown. Rather than having to go through the effort of trying to figure out the optimal layout, there exists two different tiling strategies that will produce a layout based on the data they are given.

LocalLayout

LocalLayout is the first tiling strategy that produces a layout where the grid is constructed over all of the pixels within a layer of a given tile size. The resulting layout will match the original resolution of the cells within the rasters.

Note: This layout **cannot be used for creating display layers. Rather, it is best used for layers where operations and analysis will be performed.**

```
# Creates a LocalLayout where each tile within the grid will be 256x256 pixels.
gps.LocalLayout()
```

```
# Creates a LocalLayout where each tile within the grid will be 512x512 pixels.
gps.LocalLayout(tile_size=512)
```

```
# Creates a LocalLayout where each tile within the grid will be 256x512 pixels.
gps.LocalLayout(tile_cols=256, tile_rows=512)
```

GlobalLayout

The other tiling strategy is *GlobalLayout* which makes a layout where the grid is constructed over the global extent CRS. The cell resolution of the resulting layer be multiplied by a power of 2 for the CRS. Thus, using this strategy will result in either up or down sampling of the original raster.

Note: This layout strategy **should be used when the resulting layer is to be dispalyed in a TMS server.**

```
# Creates a GobalLayout instance with the default values
gps.GlobalLayout()
```

```
# Creates a GlobalLayout instance for a zoom of 12
gps.GlobalLayout(zoom=12)
```

You may have noticed from the above two examples that `GlobalLayout` does not create layout for a given zoom level by default. Rather, it determines what the zoom should be based on the size of the cells within the rasters. If you do want to create a layout for a specific zoom level, then the `zoom` parameter must be set.

2.3.8 SpatialKey

*SpatialKey*s describe the positions of rasters within the grid of the layout. This grid is a 2D plane where the location of a raster is represented by a pair of coordinates, `col` and `row`, respectively. As its name and attributes suggest, `SpatialKey` deals solely with spatial data.

```
gps.SpatialKey(col=0, row=0)
```

2.3.9 SpaceTimeKey

Like `SpatialKeys`, *SpaceTimeKey*s describe the position of a raster in a layout. However, the grid is a 3D plane where a location of a raster is represented by a pair of coordinates, `col` and `row`, as well as a z value that represents a point in time called, `instant`. Like the `instant` in `TemporalProjectedExtent`, this is also an instance of `datetime.datetime`. Thus, `SpaceTimeKeys` deal with spatial-temporal data.

```
gps.SpaceTimeKey(col=0, row=0, instant=time)
```

2.3.10 Bounds

Bounds represents the the extent of the layout grid in terms of keys. It has both a `minKey` and a `maxKey` attributes. These can either be a `SpatialKey` or a `SpaceTimeKey` depending on the type of data within the layer. The `minKey` is the left, uppermost cell in the grid and the `maxKey` is the right, bottommost cell.

```
# Creating a Bounds from SpatialKeys

min_spatial_key = gps.SpatialKey(0, 0)
max_spatial_key = gps.SpatialKey(10, 10)

bounds = gps.Bounds(min_spatial_key, max_spatial_key)
bounds
```

```
# Creating a Bounds from SpaceTimeKeys

min_space_time_key = gps.SpaceTimeKey(0, 0, 1.0)
max_space_time_key = gps.SpaceTimeKey(10, 10, 1.0)

gps.Bounds(min_space_time_key, max_space_time_key)
```

2.3.11 Metadata

Metadata contains information of the values within a layer. This data pertains to the layout, projection, and extent of the data contained within the layer.

The below example shows how to construct `Metadata` by hand, however, this is almost never required and `Metadata` can be produced using easier means. For `RasterLayer`, one can call the method, `collect_metadata()` and `TiledRasterLayer` has the attribute, `layer_metadata`.

```
# Creates Metadata for a layer with rasters that have a cell type of int16 with the␣
↪previously defined
# bounds, crs, extent, and layout definition.
gps.Metadata(bounds=bounds,
             crs=proj4,
```

(continues on next page)

(continued from previous page)

```
                cell_type=gps.CellType.INT16.value,
                extent=extent,
                layout_definition=layout_definition)
```

2.4 Working With Layers

Before begining, all examples in this guide need the following boilerplate code:

```
curl -o /tmp/cropped.tif https://s3.amazonaws.com/geopyspark-test/example-files/
↪cropped.tif
```

```
import datetime
import numpy as np
import pyproj
import geopyspark as gps

from pyspark import SparkContext
from shapely.geometry import box, Point

conf = gps.geopyspark_conf(master="local[*]", appName="layers")
pysc = SparkContext(conf=conf)
```

2.4.1 How is Data Stored and Represented in GeoPySpark?

All data that is worked with in GeoPySpark is at some point stored within an `RDD`. Therefore, it is important to understand how GeoPySpark stores, represents, and uses these `RDD`s throughout the library.

GeoPySpark does not work with PySpark `RDD`s, but rather, uses Python classes that are wrappers for Scala classes that contain and work with a Scala `RDD`. Specifically, these wrapper classes are *`RasterLayer`* and *`TiledRasterLayer`*, which will be discussed in more detail later.

Layers Are More Than RDDs

We refer to the Python wrapper classes as layers and not `RDD`s for two reasons: first, neither `RasterLayer` or `TiledRasterLayer` actually extends PySpark's `RDD` class; but more importantly, these classes contain more information than just the `RDD`. When we refer to a "layer", we mean both the `RDD` and its attributes.

The `RDD`s contained by GeoPySpark layers contain tuples which have type `(K, V)`, where `K` represents the key, and `V` represents the value. `V` will always be a *`Tile`*, but `K` differs depending on both the wrapper class and the nature of the data itself. More on this below.

RasterLayer

The `RasterLayer` class deals with *untiled data*—that is, the elements of the layer have not been normalized into a single unified layout. Each raster element may have distinct resolutions or sizes; the extents of the constituent rasters need not follow any orderly pattern. Essentially, a `RasterLayer` stores "raw" data, and its main purpose is to act as a way station on the path to acquiring *tiled data* that adheres to a specified layout.

The `RDD`s contained by `RasterLayer` objects have key type, `K`, of either *`ProjectedExtent`* or *`TemporalProjectedExtent`*, when the layer type is `SPATIAL` or `SPACETIME`, respectively.

TiledRasterLayer

`TiledRasterLayer` is the complement to `RasterLayer` and is meant to store tiled data. Tiled data has been fitted to a certain layout, meaning that it has been regularly sampled, and it has been cut up into uniformly-sized, non-overlapping pieces that can be indexed sensibly. The benefit of having data in this state is that now it will be easy to work with. It is with this class that the user will be able to, for example, perform map algebra, create pyramids, and save the layer. See below for the definitions and specific examples of these operations.

In the case of `TiledRasterLayer`, `K` is either *`SpatialKey`* or *`SpaceTimeKey`*.

2.4.2 RasterLayer

Creating RasterLayers

There are just two ways to create a `RasterLayer`: (1) through reading GeoTiffs from the local file system, S3, or HDFS; or (2) from an existing PySpark RDD.

From PySpark RDDs

The first option is to create a `RasterLayer` from a PySpark `RDD` via the *`from_numpy_rdd()`* class method. This step can be a bit more involved, as it requires the data within the PySpark RDD to be formatted in a specific way (see *How is Data Stored and Represented in GeoPySpark* for more information).

The following example constructs an `RDD` from a tuple. The first element is a `ProjectedExtent` because we have decided to make the data spatial. If we were dealing with spatial-temproal data, then `TemporalProjectedExtent` would be the first element. A `Tile` will always be the second element of the tuple.

```
arr = np.ones((1, 16, 16), dtype='int')
tile = gps.Tile.from_numpy_array(numpy_array=np.array(arr), no_data_value=-500)

extent = gps.Extent(0.0, 1.0, 2.0, 3.0)
projected_extent = gps.ProjectedExtent(extent=extent, epsg=3857)

rdd = pysc.parallelize([(projected_extent, tile), (projected_extent, tile)])
multiband_raster_layer = gps.RasterLayer.from_numpy_rdd(layer_type=gps.LayerType.
↪SPATIAL, numpy_rdd=rdd)
multiband_raster_layer
```

From GeoTiffs

The *`get()`* function in the `geopyspark.geotrellis.geotiff` module creates an instance of `RasterLayer` from GeoTiffs. These files can be located on either your local file system, HDFS, or S3. In this example, a GeoTiff with spatial data is read locally.

```
raster_layer = gps.geotiff.get(layer_type=gps.LayerType.SPATIAL, uri="file:///tmp/
↪cropped.tif")
raster_layer
```

Using RasterLayer

This next section goes over the methods of `RasterLayer`. It should be noted that not all methods contained within this class will be covered. More information on the methods that deal with the visualization of the contents of the layer can be found in the *Visualizing Data in GeoPySpark*.

Converting to a Python RDD

By using `to_numpy_rdd()`, the base `RasterLayer` will be serialized into a Python `RDD`. This will convert all of the first values within each tuple to either `ProjectedExtent` or `TemporalProjectedExtent`, and the second value to `Tile`.

```
python_rdd = raster_layer.to_numpy_rdd()
python_rdd
```

```
python_rdd.first()
```

SpaceTime Layer to Spatial Layer

If you're working with a spatial-temporal layer and would like to convert it to a spatial layer, then you can use the `to_spatial_layer`()` method. This changes the keys of the `RDD` within the layer by converting `TemporalProjectedExtent` to `ProjectedExtent`.

```
# Creating the space time layer

instant = datetime.datetime.now()
temporal_projected_extent = gps.TemporalProjectedExtent(extent=projected_extent.
↪extent,
                                                        epsg=projected_extent.epsg,
                                                        instant=instant)

space_time_rdd = pysc.parallelize([temporal_projected_extent, tile])
space_time_layer = gps.RasterLayer.from_numpy_rdd(layer_type=gps.LayerType.SPACETIME,␣
↪numpy_rdd=space_time_rdd)
space_time_layer
```

```
# Converting the SpaceTime layer to a Spatial layer

space_time_layer.to_spatial_layer()
```

Collecting Metadata

The `Metadata` of a layer contains information of the values within it. This data pertains to the layout, projection, and extent of the data found within the layer.

`collect_metadata()` will return the `Metadata` of the layer that fits the `layout` given.

```
# Collecting Metadata with the default LocalLayout()
metadata = raster_layer.collect_metadata()
metadata
```

```
# Collecting Metadata with the default GlobalLayout()
raster_layer.collect_metadata(layout=gps.GlobalLayout())
```

```
# Collecting Metadata with a LayoutDefinition
extent = gps.Extent(0.0, 0.0, 33.0, 33.0)
tile_layout = gps.TileLayout(2, 2, 256, 256)
layout_definition = gps.LayoutDefinition(extent, tile_layout)

raster_layer.collect_metadata(layout=layout_definition)
```

Reproject

`reproject()` will change the projection of the rasters within the layer to the given `target_crs`. This method does not sample past the tiles' boundaries.

```
# The CRS of the layer before reprojecting
metadata.crs
```

```
# The CRS of the layer after reprojecting
raster_layer.reproject(target_crs=3857).collect_metadata().crs
```

Tiling Data to a Layout

`tile_to_layout()` will tile and format the rasters within a `RasterLayer` to a given layout. The result of this tiling is a new instance of `TiledRasterLayer`. This output contains the same data as its source `RasterLayer`, however, the information contained within it will now be orginized according to the given layout.

During this step it is also possible to reproject the `RasterLayer`. This can be done by specifying the `target_crs` to reproject to. Reprojecting using this method produces a different result than what is returned by the `reproject` method. Whereas the latter does not sample past the boundaries of rasters within the layer, the former does. This is important as anything with a `GlobalLayout` needs to sample past the boundaries of the rasters.

From Metadata

Create a `TiledRasterLayer` that contains the layout from the given `Metadata`.

Note: If the specified `target_crs` is different from what's in the metadata, then an error will be thrown.

```
raster_layer.tile_to_layout(layout=metadata)
```

From LayoutDefinition

```
raster_layer.tile_to_layout(layout=layout_definition)
```

From LocalLayout

```
raster_layer.tile_to_layout(gps.LocalLayout())
```

From GlobalLayout

```
tiled_raster_layer = raster_layer.tile_to_layout(gps.GlobalLayout())
tiled_raster_layer
```

From A TiledRasterLayer

One can tile a `RasterLayer` to the same layout as a `TiledRasterLayout`.

Note: If the specifying `target_crs` is different from the other layer's, then an error will be thrown.

```
raster_layer.tile_to_layout(layout=tiled_raster_layer)
```

2.4.3 TiledRasterLayer

Creating TiledRasterLayers

For this guide, we will just go over one initialization method for `TiledRasterLayer`, `from_numpy_rdd`. However, there are other ways to create this class. These additional creation strategies can be found in the [map algebra guide].

From PySpark RDD

Like `RasterLayers`, `TiledRasterLayers` can be created from `RDDs` using *`from_numpy_rdd()`*. What is different, however, is that *`Metadata`* must also be passed in during initialization. This makes creating `TiledRasterLayers` this way a little bit more arduous.

The following example constructs an `RDD` from a tuple. The first element is a `SpatialKey` because we have decided to make the data spatial. See *How is Data Stored and Represented in GeoPySpark* for more information.

```
data = np.zeros((1, 512, 512), dtype='float32')
tile = gps.Tile.from_numpy_array(numpy_array=data, no_data_value=-1.0)
instant = datetime.datetime.now()

layer = [(gps.SpaceTimeKey(row=0, col=0, instant=instant), tile),
         (gps.SpaceTimeKey(row=1, col=0, instant=instant), tile),
         (gps.SpaceTimeKey(row=0, col=1, instant=instant), tile),
         (gps.SpaceTimeKey(row=1, col=1, instant=instant), tile)]

rdd = pysc.parallelize(layer)

extent = gps.Extent(0.0, 0.0, 33.0, 33.0)
layout = gps.TileLayout(2, 2, 512, 512)
bounds = gps.Bounds(gps.SpaceTimeKey(col=0, row=0, instant=instant), gps.
↪SpaceTimeKey(col=1, row=1, instant=instant))
layout_definition = gps.LayoutDefinition(extent, layout)

metadata = gps.Metadata(
    bounds=bounds,
    crs='+proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +a=6378137 +b=6378137 +towgs84=0,0,0,
↪0,0,0,0 +units=m +no_defs ',
    cell_type='float32ud-1.0',
```

(continues on next page)

(continued from previous page)

```
    extent=extent,
    layout_definition=layout_definition)

space_time_tiled_layer = gps.TiledRasterLayer.from_numpy_rdd(layer_type=gps.LayerType.
↪SPACETIME,
                                                             numpy_rdd=rdd,␣
↪metadata=metadata)
space_time_tiled_layer
```

Using TiledRasterLayers

This section will go over the methods found within `TiledRasterLayer`. Like with `RasterLayer`, not all methods within this class will be covered in this guide. More information on the methods that deal with the visualization of the contents of the layer can be found in *Visualizing Data in GeoPySpark*; and those that deal with map algebra can be found in the [map algebra guide].

Converting to a Python RDD

By using `to_numpy_rdd()`, the base `TiledRasterLayer` will be serialized into a Python `RDD`. This will convert all of the first values within each tuple to either `SpatialKey` or `SpaceTimeKey`, and the second value to `Tile`.

```
python_rdd = tiled_raster_layer.to_numpy_rdd()
```

```
python_rdd.first()
```

SpaceTime Layer to Spatial Layer

If you're working with a spatiotemporal layer and would like to convert it to a spatial layer, then you can use the `to_spatial_layer()` method. This changes the keys of the `RDD` within the layer by converting `SpaceTimeKey` to `SpatialKey`.

```
# Converting the SpaceTime layer to a Spatial layer

space_time_tiled_layer.to_spatial_layer()
```

Repartitioning

While not an `RDD`, `TiledRasterLayer` does contain an underlying `RDD`, and thus, it can be repartitioned using the `repartition()` method.

```
# Repartition the internal RDD to have 120 partitions
tiled_raster_layer.repartition(num_partitions=120)
```

Lookup

If there is a particular tile within the layer that is of interest, it is possible to retrieve it as a `Tile` using the `lookup()` method.

```
min_key = tiled_raster_layer.layer_metadata.bounds.minKey

# Retrieve the Tile that is located at the smallest column and row of the layer
tiled_raster_layer.lookup(col=min_key.col, row=min_key.row)
```

Masking

By using `mask()` method, the `TiledRasterRDD` can be masekd using one or more Shapely geometries.

```
layer_extent = tiled_raster_layer.layer_metadata.extent

# Polygon to mask a region of the layer
mask = box(layer_extent.xmin,
           layer_extent.ymin,
           layer_extent.xmin + 20,
           layer_extent.ymin + 20)

tiled_raster_layer.mask(geometries=mask)
```

```
mask_2 = box(layer_extent.xmin + 50,
             layer_extent.ymin + 50,
             layer_extent.xmax - 20,
             layer_extent.ymax - 20)

# Multiple Polygons can be given to mask the layer
tiled_raster_layer.mask(geometries=[mask, mask_2])
```

Normalize

`normalize()` will linearly transform the data within the layer such that all values fall within a given range.

```
# Normalizes the layer so that the new min value is 0 and the new max value is 60000
tiled_raster_layer.normalize(new_min=0, new_max=60000)
```

Pyramiding

When using a layer for a TMS server, it is important that the layer is pyramided. That is, we create a level-of-detail hierarchy that covers the same geographical extent, while each level of the pyramid uses one quarter as many pixels as the next level. This allows us to zoom in and out when the layer is being displayed without using extraneous detail. The `pyramid()` method will produce an instance of `Pyramid` that will contain within it multiple `TiledRasterLayers`. Each layer corresponds to a zoom level, and the number of levels depends on the `zoom_level` of the source layer. With the max zoom of the `Pyramid` being the source layer's `zoom_level`, and the lowest zoom being 0.

For more information on the `Pyramid` class, see the *Pyramid* section of the visualization guide.

```
# This creates a Pyramid with zoom levels that go from 0 to 11 for a total of 12.
tiled_raster_layer.pyramid()
```

Reproject

This is similar to the `reproject` method for `RasterLayer` where the reprojection will not sample past the tiles' boundaries. This means the layout of the tiles will be changed so that they will take on a `LocalLayout` rather than a `GlobalLayout` (read more about these layouts here). Because of this, whatever `zoom_level` the `TiledRasterLayer` has will be changed to 0 since the area being represented changes to just the tiles.

```
# The zoom_level and crs of the TiledRasterLayer before reprojecting
tiled_raster_layer.zoom_level, tiled_raster_layer.layer_metadata.crs
```

```
reprojected_tiled_raster_layer = tiled_raster_layer.reproject(target_crs=3857)

# The zoom_level and crs of the TiledRasterLayer after reprojecting
reprojected_tiled_raster_layer.zoom_level, reprojected_tiled_raster_layer.layer_
↪metadata.crs
```

Stitching

Using *stitch()* will produce a single `Tile` by stitching together all of the tiles within the `TiledRasterLayer`. This can only be done with spatial layers, and is not recommended if the data contained within the layer is large, as it can cause a crash due to the size of the resulting `Tile`.

```
# Creates a Tile with an underlying numpy array with a size of (1, 6144, 1536).
tiled_raster_layer.stitch().cells.shape
```

Saving a Stitched Layer

The *save_stitched()* method both stitches and saves a layer as a GeoTiff.

```
# Saves the stitched layer to /tmp/stitched.tif
tiled_raster_layer.save_stitched(path='/tmp/stitched.tif')
```

It is also possible to specify the regions of layer to be saved when it is stitched.

```
layer_extent = tiled_raster_layer.layer_metadata.layout_definition.extent

# Only a portion of the stitched layer needs to be saved, so we will create a sub␣
↪Extent to crop to.
sub_exent = gps.Extent(xmin=layer_extent.xmin + 10,
                       ymin=layer_extent.ymin + 10,
                       xmax=layer_extent.xmax - 10,
                       ymax=layer_extent.ymax - 10)

tiled_raster_layer.save_stitched(path='/tmp/cropped-stitched.tif', crop_bounds=sub_
↪exent)
```

```
# In addition to the sub Extent, one can also choose how many cols and rows will be␣
↪in the saved in the GeoTiff.
tiled_raster_layer.save_stitched(path='/tmp/cropped-stitched-2.tif',
                                 crop_bounds=sub_exent,
                                 crop_dimensions=(1000, 1000))
```

Tiling Data to a Layout

This is similar to RasterLayer's tile_to_layout method, except for one important detail. If performing a tile_to_layout() on a TiledRasterLayer that contains a zoom_level, that zoom_level could be lost or changed depending on the layout and/or target_crs chosen. Thus, it is important to keep that in mind in retiling a TiledRasterLayer.

```
# Original zoom_level of the source TiledRasterLayer
tiled_raster_layer.zoom_level
```

```
# zoom_level will be lost in the resulting TiledRasterlayer
tiled_raster_layer.tile_to_layout(layout=gps.LocalLayout())
```

```
# zoom_level will be changed in the resulting TiledRasterLayer
tiled_raster_layer.tile_to_layout(layout=gps.GlobalLayout(), target_crs=3857)
```

```
# zoom_level will reamin the same in the resulting TiledRasterLayer
tiled_raster_layer.tile_to_layout(layout=gps.GlobalLayout(zoom=11))
```

Getting Point Values

get_point_values() takes a collection of shapely.geometry.Points and returns the value(s) that are at the given point in the layer. The number of values returned depends on the number of bands the values have, as there will be one value per band.

It is also possible to pass in a ResampleMethod to this method, but not all are supported. The following are all of the ResampleMethods that can be used to calculate point values:

- ResampleMethod.NEAREST_NEIGHBOR
- ResampleMethod.BILINEAR
- ResampleMethod.CUBIC_CONVOLUTION
- ResampleMethod.CUBIC_SPLINE

Getting the Point Values From a SPATIAL Layer

When using get_point_values on a layer with a LayerType of SPATIAL, the results will be paired as (shapely.geometry.Point, [float]). Where each given Point will be paired with the values it intersects.

```
# Creating the points
extent = tiled_raster_layer.layer_metadata.extent

p1 = Point(extent.xmin, extent.ymin + 0.5)
p2 = Point(extent.xmax , extent.ymax - 1.0)
```

Giving a [shapely.geometry.Point] to get_point_values

When points is given as a [shapely.geometry.Point], then the ouput will be a [(shapely.geometry.Point, [float])].

```
tiled_raster_layer.get_point_values(points=[p1, p2])
```

Giving a {k: shapely.geometry.Point} to get_point_values

When `points` is given as a `{k: shapely.geometry.Point}`, then the ouput will be a `{k: (shapely.geometry.Point, [float])}`.

```
tiled_raster_layer.get_point_values(points={'point 1': p1, 'point 2': p2})
```

Getting the Point Values From a SPACETIME Layer

When using `get_point_values` on a layer with a `LayerType` of `SPACETIME`, the results will be paired as `(shapely.geometry.Point, [(datetime.datetime, [float])])`. Where each given `Point` will be paired with a list of tuples that contain the values it intersects and those values' corresponding timestamps.

```
st_extent = space_time_tiled_layer.layer_metadata.extent

p1 = Point(st_extent.xmin, st_extent.ymin + 0.5)
p2 = Point(st_extent.xmax , st_extent.ymax - 1.0)
```

Giving a [shapely.geometry.Point] to get_point_values

When `points` is given as a `[shapely.geometry.Point]`, then the ouput will be a `[(shapely.geometry.Point, [(datetime.datetime, [float])])]`.

```
space_time_tiled_layer.get_point_values(points=[p1, p2])
```

Giving a {k: shapely.geometry.Point} to get_point_values

When `points` is given as a `{k: shapely.geometry.Point}`, then the ouput will be a `{k: (shapely.geometry.Point, [(datetime.datetime, [float])])}`.

```
space_time_tiled_layer.get_point_values(points={'point 1': p1, 'point 2': p2})
```

Aggregating the Values of Each Cell

`aggregate_by_cell()` will compute an aggregate summary for each cell of all values for each key. Thus, if there are multiple copies of the same key in the layer, then the resulting layer will contain just a single instance of that key with its corresponding value being the aggregate summary of all the values that share that key.

Not all `Operations` are supported. The following ones can be used in `aggregate_by_cell`:

- `Operation.SUM`
- `Operation.MIN`
- `Operation.MAX`
- `Operation.MEAN`

- Operation.VARIANCE
- Operation.STANDARD_DEVIATION

```
unioned_layer = gps.union(layers=[tiled_raster_layer, tiled_raster_layer + 1])

# Sum the values of the unioned_layer
unioned_layer.aggregate_by_cell(operation=gps.Operation.SUM)

# Get the max value for each cell
unioned_layer.aggregate_by_cell(operation=gps.Operation.MAX)
```

2.4.4 General Methods

There exist methods that are found in both RasterLayer and TiledRasterLayer. These methods tend to perform more general analysis/tasks, thus making them suitable for both classes. This next section will go over these methods.

Note: In the following examples, both RasterLayers and TiledRasterLayers will be used. However, they can easily be subsituted with the other class.

Unioning Layers Togther

To combine the contents of multiple layers together, one can use the *union()* method. This will produce either a new RasterLayer or TiledRasterLayer that contains all of the elements from the given layers.

Note: The resulting layer can contain duplicate keys.

```
gps.union(layers=[tiled_raster_layer, tiled_raster_layer])
```

Selecting a SubSection of Bands

To select certain bands to work with, the bands method will take either a single or collection of band indices and will return the subset as a new RasterLayer or TiledRasterLayer.

Note: There could high performance costs if operations are performed between two sub-bands of a large dataset. Thus, if you're working with a large amount of data, then it is recommended to do band selection before reading them in.

```
# Selecting the second band from the layer
multiband_raster_layer.bands(1)
```

```
# Selecting the first and second bands from the layer
multiband_raster_layer.bands([0, 1])
```

Combining Bands of Two Or More Layers

The *combine_bands()* method will concatenate the bands of values that share a key between two or more layers. Thus, the resulting layer will contain a new Tile for each shared key where the Tile will contain all of the bands from the given layers.

The order in which the layers are passed into combine_bands matters. Where the resulting values' bands will be ordered based on their position of their respective layer.

```
# Setting up example RDD
twos = np.ones((1, 16, 16), dtype='int') + 1
twos_tile = gps.Tile.from_numpy_array(numpy_array=np.array(twos), no_data_value=-500)

twos_rdd = pysc.parallelize([(projected_extent, twos_tile)])
twos_raster_layer = gps.RasterLayer.from_numpy_rdd(layer_type=gps.LayerType.SPATIAL,␣
↪numpy_rdd=twos_rdd)
```

```
# The resulting values of the layer will have 2 bands: the first will be all ones,
# and the last band will be all twos
gps.combine_bands(layers=[multiband_raster_layer, twos_raster_layer])
```

```
# The resulting values of the layer will have 2 bands: the first will be all twos and␣
↪the
# other band will be all ones
gps.combine_bands(layers=[twos_raster_layer, multiband_raster_layer])
```

Collecting the Keys of a Layer

To collect all of the keys of a layer, use the `collect_keys` method.

```
# Returns a list of ProjectedExtents
multiband_raster_layer.collect_keys()

# Returns a list of a SpatialKeys
tiled_raster_layer.collect_keys()

# Returns a list of SpaceTimeKeys
space_time_tiled_layer.collect_keys()
```

Filtering a Layer By Times

Using the `filter_by_times` method will produce a layer whose values fall within the given time interval(s).

Filtering By a Single Instant

A single `datetime.datetime` instance can be used to filter the layer. If that is the case then only exact matches with the given time will be kept.

```
space_time_layer.filter_by_times(time_intervals=[instant])
```

Filtering By Intervals

Various time intervals can also be given as well, and any keys whose `instant` falls within the time spans will be kept in the layer.

```
end_date_1 = instant + datetime.timedelta(days=3)
end_date_2 = instant + datetime.timedelta(days=5)

# Will filter out any value whose key does not fall in the range of
```

(continues on next page)

(continued from previous page)

```
# instant and end_date_1
space_time_layer.filter_by_times(time_intervals=[instant, end_date_1])

# Will filter out any value whose key does not fall in the range of
# instant and end_date_1 OR whose key does not match end_date_2
space_time_layer.filter_by_times(time_intervals=[instant, end_date_1, end_date_2])
```

Converting the Data Type of the Rasters' Cells

The `convert_data_type` method will convert the types of the cells within the rasters of the layer to a new data type. The `noData` value can also be set during this conversion, and if it's not set, then there will be no `noData` value for the resulting rasters.

```
# The data type of the cells before converting
metadata.cell_type
```

```
# Changing the cell type to int8 with a noData value of -100.
raster_layer.convert_data_type(new_type=gps.CellType.INT8, no_data_value=-100).
↪collect_metadata().cell_type
```

```
# Changing the cell type to int32 with no noData value.
raster_layer.convert_data_type(new_type=gps.CellType.INT32).collect_metadata().cell_
↪type
```

Reclassify Cell Values

`reclassify` changes the cell values based on the `value_map` and `classification_strategy` given. In addition to these two parameters, the `data_type` of the cells also needs to be given. This is either `int` or `float`.

```
# Values of the first tile before being reclassified
multiband_raster_layer.to_numpy_rdd().first()[1]
```

```
# Change all values greater than or equal to 1 to 10
reclassified = multiband_raster_layer.reclassify(value_map={1: 10},
                                                 data_type=int,
                                                 classification_strategy=gps.
↪ClassificationStrategy.GREATER_THAN_OR_EQUAL_TO)
reclassified.to_numpy_rdd().first()[1]
```

Merging the Values of a Layer Together

By using the `merge` method, all values that share a key within the layer will be merged together to form a new, single value. This is accomplished by replacing the cells of one value with another's. However, not all cells, if any, may be replaced. When merging the cell of values, the following steps are taken to determine if a cell's value should be changed:

1. If the cell contains a `NoData` value, then it will be replaced.
2. If no `NoData` value is set, then a cell with a vlue of 0 will be replaced.
3. if neither of the above are true, then the cell retains its value.

```python
# Creating the layers
no_data = np.full((1, 4, 4), -1)
zeros = np.zeros((1, 4, 4))

def create_layer(no_data_value=None):
    data_tile = gps.Tile.from_numpy_array(numpy_array=no_data, no_data_value=no_data_
↪value)
    zeros_tile = gps.Tile.from_numpy_array(numpy_array=zeros, no_data_value=no_data_
↪value)

    layer_rdd = pysc.parallelize([(projected_extent, data_tile), (projected_extent,␣
↪zeros_tile)])
    return gps.RasterLayer.from_numpy_rdd(layer_type=gps.LayerType.SPATIAL, numpy_
↪rdd=layer_rdd)

# Resulting layer has a no_data_value of -1
no_data_layer = create_layer(-1)

# Resutling layer has no no_data_value
no_no_data_layer = create_layer()
```

```python
# The resulting merged value will be all zeros since -1 is the noData value
no_data_layer.merge()

# The resulting merged value will be all -1's as ``no_data_value`` was set.
no_no_data_layer.merge()
```

Mapping Over the Cells

It is possible to work with the cells within a layer directly via the `map_cells` method. This method takes a function that expects a numpy array and a noData value as parameters, and returns a new numpy array. Thus, the function given would have the following type signature:

```python
def input_function(numpy_array: np.ndarray, no_data_value=None) -> np.ndarray
```

The given function is then applied to each `Tile` in the layer.

Note: In order for this method to operate, the internal `RDD` first needs to be deserialized from Scala to Python and then serialized from Python back to Scala. Because of this, it is recommended to chain together all functions to avoid unnecessary serialization overhead.

```python
def add_one(cells, _):
    return cells + 1

# Mapping with a single funciton
raster_layer.map_cells(add_one)
```

```python
def divide_two(cells, _):
    return (add_one(cells) / 2)

# Chaning together two functions to be mapped
raster_layer.map_cells(divide_two)
```

Mapping Over Tiles

Like map_cells, map_tiles maps a given function over all of the Tiles within the layer. It takes a function that expects a Tile and returns a Tile. Therefore, the input function's type signature would be this:

```
def input_function(tile: Tile) -> Tile
```

Note: In order for this method to operate, the internal RDD first needs to be deserialized from Scala to Python and then serialized from Python back to Scala. Because of this, it is recommended to chain together all functions to avoid unnecessary serialization overhead.

```
def minus_two(tile):
    return gps.Tile.from_numpy_array(tile.cells - 2, no_data_value=tile.no_data_value)

raster_layer.map_tiles(minus_two)
```

Calculating the Histogram for the Layer

It is possible to calculate the histogram of a layer either by using the get_histogram or the get_class_histogram method. Both of these methods produce a Histogram, however, the way the data is represented within the resulting histogram differs depending on the method used. get_histogram will produce a histogram whose values are floats. Whereas get_class_histogram returns a histogram whose values are ints.

For more informaiton on the Histogram class, please see the Histogram [guide].

```
# Returns a Histogram whose underlying values are floats
tiled_raster_layer.get_histogram()
```

```
# Returns a Histogram whose underlying values are ints
tiled_raster_layer.get_class_histogram()
```

Finding the Quantile Breaks for the Layer

If you wish to find the quantile breaks for a layer without a Histogram, then you can use the get_quantile_breaks method.

```
tiled_raster_layer.get_quantile_breaks(num_breaks=3)
```

Quantile Breaks for Exact Ints

There is another version of get_quantile_breaks called get_quantile_breaks_exact_int that will count exact integer values. However, if there are too many values within the layer, then memory errors could occur.

```
tiled_raster_layer.get_quantile_breaks_exact_int(num_breaks=3)
```

Finding the Min and Max Values of a Layer

The get_min_max method will find the min and max value for the layer. The result will always be (float, float) regardless of the data type of the cells.

```
tiled_raster_layer.get_min_max()
```

Converting the Values of a Layer to PNGs

Via the `to_png_rdd` method, one can convert each value within a layer to a PNG in the form of `bytes`. In order to convert each value to a PNG, one needs to supply a `ColorMap`. For more information on the `ColorMap` class, please see the *ColorMap* section of the docs.

In addition to converting each value to a PNG, the resulting collection of `(K, V)`s will be held in a Python `RDD`.

```
hist = tiled_raster_layer.get_histogram()
cmap = gps.ColorMap.build(hist, 'viridis')

tiled_raster_layer.to_png_rdd(color_map=cmap)
```

Converting the Values of a Layer to GeoTiffs

Similar to `to_png_rdd`, only `to_geotiff_rdd` will return a Python `RDD[(K, bytes)]` where the `bytes` represent a GeoTiff.

Selecting a StorageMethod

There are two different ways the segments of a GeoTiff can be formatted: `StorageMethod.STRIPED` or `StorageMethod.TILED`. This is represented by the `storage_method` parameter. By default, `StorageMethod.STRIPED` is used.

Selecting the Size of the Segments

There are two different parameters that control the size of each segment: `rows_per_strip` and `tile_dimensions`. Only one of these values needs to be set, and that is determined by what the `storage_method` is.

If the `storage_method` is `StorageMethod.STRIPED`, then `rows_per_strip` will be the parameter to change. By default, the `rows_per_strip` will be calculated so that each strip is 8K or less.

If the `storage_method` is `StorageMethod.TILED`, then `tile_dimensions` can be set. This is given as a `(int, int)` where the first value is the number of `cols` and the second is the number of `` rows` ``. By default, the `tile_dimensions` is `(256, 256)`.

Selecting a CompressionMethod

The two types of compressions that can be chosen are: `Compression.NO_COMPRESSION` or `Compression.DEFLATE_COMPRESSION`. By default, the `compression` parameter is set to `Compression.NO_COMPRESSION`.

Selecting a ColorSpace

The `color_space` parameter determines how the colors should be organized in each GeoTiff. By default, it's `ColorSpace.BLACK_IS_ZERO`.

Passing in a ColorMap

A `ColorMap` instance can be passed in so that the resulting GeoTiffs are in a different gradiant. By default, `color_map` is `None`. To learn more about `ColorMap`, see the *ColorMap* section of the docs.

```
# Creates an RDD[(K, bytes)] with the default parameters
tiled_raster_layer.to_geotiff_rdd()

# Creates an RDD whose GeoTiffs are tiled with a size of (128, 128)
tiled_raster_layer.to_geotiff_rdd(storage_method=gps.StorageMethod.TILED, tile_
↪dimensions=(128, 128))
```

2.4.5 RDD Methods

As mentioned in the section on `TiledRasterLayer`'s *repartition method*, `TiledRasterLayer` has methods to work with its internal `RDD`. This holds true for `RasterLayer` as well.

The following is a list of `RDD` with examples that are supported by both classes.

Cache

```
raster_layer.cache()
```

Persist

```
# If no level is given, then MEMORY_ONLY will be used
tiled_raster_layer.persist()
```

Unpersist

```
tiled_raster_layer.unpersist()
```

getNumberOfPartitions

```
raster_layer.getNumPartitions()
```

Count

```
raster_layer.count()
```

isEmpty

```
raster_layer.isEmpty()
```

2.5 Catalog

The `catalog` module allows for users to retrieve information, query, and write to/from GeoTrellis layers.

Before begining, all examples in this guide need the following boilerplate code:

```
curl -o /tmp/cropped.tif https://s3.amazonaws.com/geopyspark-test/example-files/
↪cropped.tif
```

```
import datetime
import geopyspark as gps
import numpy as np

from pyspark import SparkContext
from shapely.geometry import MultiPolygon, box

conf = gps.geopyspark_conf(master="local[*]", appName="layers")
pysc = SparkContext(conf=conf)

# Setting up the Spatial Data to be used in this example

spatial_raster_layer = gps.geotiff.get(layer_type=gps.LayerType.SPATIAL, uri="/tmp/
↪cropped.tif")
spatial_tiled_layer = spatial_raster_layer.tile_to_layout(layout=gps.GlobalLayout(),␣
↪target_crs=3857)

# Setting up the Spatial-Temporal Data to be used in this example

def make_raster(x, y, v, cols=4, rows=4, crs=4326):
    cells = np.zeros((1, rows, cols), dtype='float32')
    cells.fill(v)
    # extent of a single cell is 1
    extent = gps.TemporalProjectedExtent(extent = gps.Extent(x, y, x + cols, y +␣
↪rows),
                                         epsg=crs,
                                         instant=datetime.datetime.now())

    return (extent, gps.Tile.from_numpy_array(cells))

layer = [
    make_raster(0, 0, v=1),
    make_raster(3, 2, v=2),
    make_raster(6, 0, v=3)
]

rdd = pysc.parallelize(layer)
space_time_raster_layer = gps.RasterLayer.from_numpy_rdd(gps.LayerType.SPACETIME, rdd)
space_time_tiled_layer = space_time_raster_layer.tile_to_layout(layout=gps.
↪GlobalLayout(tile_size=5))
space_time_pyramid = space_time_tiled_layer.pyramid()
```

2.5.1 What is a Catalog?

A catalog is a directory where saved layers and their attributes are organized and stored in a certain manner. Within a catalog, there can exist multiple layers from different data sets. Each of these layers, in turn, are their own directories which contain two folders: one where the data is stored and the other for the metadata. The data for each layer is broken up into zoom levels and each level has its own folder within the data folder of the layer. As for the metadata, it is also broken up by zoom level and is stored as `json` files within the metadata folder.

Here's an example directory structure of a catalog:

```
layer_catalog/
  layer_a/
    metadata_for_layer_a/
      metadata_layer_a_zoom_0.json
      ....
    data_for_layer_a/
      0/
        data
        ...
      1/
        data
        ...
      ...
  layer_b/
  ...
```

2.5.2 Accessing Data

GeoPySpark supports a number of different backends to save and read information from. These are the currently supported backends:

- LocalFileSystem
- HDFS
- S3
- Cassandra
- HBase
- Accumulo

Each of these needs to be accessed via the `URI` for the given system. Here are example `URIs` for each:

- **Local Filesystem**: file://my_folder/my_catalog/
- **HDFS**: hdfs://my_folder/my_catalog/
- **S3**: s3://my_bucket/my_catalog/
- **Cassandra**: cassandra://[user:password@]zookeeper[:port][/keyspace][?attributes=table1[&layers=table2]]
- **HBase**: hbase://zookeeper[:port][?master=host][?attributes=table1[&layers=table2]]
- **Accumulo**: accumulo://[user[:password]@]zookeeper/instance-name[?attributes=table1[&layers=table2]]

It is important to note that neither HBase nor Accumulo have native support for `URIs`. Thus, GeoPySpark uses its own pattern for these two systems.

A Note on Formatting Tiles

A small, but important, note needs to be made about how tiles that are saved and/or read in are formatted in GeoPySpark. All tiles will be treated as a `MultibandTile`. Regardless if they were one to begin with. This was a design choice that was made to simplify both the backend and the API of GeoPySpark.

2.5.3 Saving Data to a Backend

The *write()* function will save a given *TiledRasterLayer* to a specified backend. If the catalog does not exist when calling this function, then it will be created along with the saved layer.

Note: It is not possible to save a layer to a catalog if the layer name and zoom already exist. If you wish to overwrite an existing, saved layer then it must be deleted before writing the new one.

Note: Saving a `TiledRasterLayer` that does not have a `zoom_level` will save the layer to a zoom of 0. Thus, when it is read back out from the catalog, the resulting `TiledRasterLayer` will have a `zoom_level` of 0.

Saving a Spatial Layer

Saving a spatial layer is a straight forward task. All that needs to be supplied is a `URI`, the name of the layer, and the layer to be saved.

```
# The zoom level which will be saved
spatial_tiled_layer.zoom_level
```

```
# This will create a catalog called, "spatial-catalog" in the /tmp directory.
# Within it, a layer named, "spatial-layer" will be saved.
gps.write(uri='file:///tmp/spatial-catalog', layer_name='spatial-layer', tiled_raster_
↪layer=spatial_tiled_layer)
```

Saving a Spatial Temporal Layer

When saving a spatial-temporal layer, one needs to consider how the records within the catalog will be spaced; which in turn, determines the resolution of index. The `TimeUnit` enum class contains all available units of time that can be used to space apart data in the catalog.

```
# The zoom level which will be saved
space_time_tiled_layer.zoom_level
```

```
# This will create a catalog called, "spacetime-catalog" in the /tmp directory.
# Within it, a layer named, "spacetime-layer" will be saved and each indice will be␣
↪spaced apart by SECONDS
gps.write(uri='file:///tmp/spacetime-catalog',
          layer_name='spacetime-layer',
          tiled_raster_layer=space_time_tiled_layer,
          time_unit=gps.TimeUnit.SECONDS)
```

Saving a Pyramid

For those that are unfamiliar with the `Pyramid` class, please see the *Pyramid* section of the visualization guide. Otherwise, please continue on.

As of right now, there is no way to directly save a `Pyramid`. However, because a `Pyramid` is just a collection of `TiledRasterLayers` of different zooms, it is possible to iterate through the layers of the `Pyramid` and save one individually.

```
for zoom, layer in space_time_pyramid.levels.items():
    # Because we've already written a layer of the same name to the same catalog with␣
↪a zoom level of 7,
```

(continues on next page)

(continued from previous page)

```
    # we will skip writing the level 7 layer.
    if zoom != 7:
        gps.write(uri='file:///tmp/spacetime-catalog',
                  layer_name='spacetime-layer',
                  tiled_raster_layer=layer,
                  time_unit=gps.TimeUnit.SECONDS)
```

2.5.4 Reading Metadata From a Saved Layer

It is possible to retrieve the *Metadata* for a layer without reading in the whole layer. This is done using the *read_layer_metadata()* function. There is no difference between spatial and spatial-temporal layers when using this function.

```
# Metadata from the TiledRasterLayer
spatial_tiled_layer.layer_metadata
```

```
# Reads the Metadata from the spatial-layer of the spatial-catalog for zoom level 11
gps.read_layer_metadata(uri="file:///tmp/spatial-catalog",
                        layer_name="spatial-layer",
                        layer_zoom=11)
```

2.5.5 Reading a Tile From a Saved Layer

One can read a single tile that has been saved to a layer using the *read_value()* function. This will either return a *Tile* or `None` depending on whether or not the specified tile exists.

Reading a Tile From a Saved, Spatial Layer

```
# The Tile being read will be the smallest key of the layer
min_key = spatial_tiled_layer.layer_metadata.bounds.minKey

gps.read_value(uri="file:///tmp/spatial-catalog",
               layer_name="spatial-layer",
               layer_zoom=11,
               col=min_key.col,
               row=min_key.row)
```

Reading a Tile From a Saved, Spatial-Temporal Layer

```
# The Tile being read will be the largest key of the layer
max_key = space_time_tiled_layer.layer_metadata.bounds.maxKey

gps.read_value(uri="file:///tmp/spacetime-catalog",
               layer_name="spacetime-layer",
               layer_zoom=7,
               col=max_key.col,
               row=max_key.row,
               zdt=max_key.instant)
```

2.5.6 Reading a Layer

There are two ways one can read a layer in GeoPySpark: reading the entire layer or just portions of it. The former will be the goal discussed in this section. While all of the layer will be read, the function for doing so is called, *query()*. There is no difference between spatial and spatial-temporal layers when using this function.

Note: What distinguishes between a full and partial read is the parameters given to `query`. If no filters were given, then the whole layer is read.

```
# Returns the entire layer that was at zoom level 11.
gps.query(uri="file:///tmp/spatial-catalog",
          layer_name="spatial-layer",
          layer_zoom=11)
```

2.5.7 Querying a Layer

When only a certain section of the layer is of interest, one can retrieve these areas of the layer through the `query` method. The resulting `TiledRasterLayer` will contain all of the `Tiles` that the queried intersects, not just the area itself.

Depending on the type of data being queried, there are a couple of ways to filter what will be returned.

Querying a Spatial Layer

One can query an area of a spatial layer that covers the region of interest by providing a geometry that represents this region. This area can be represented as: `shapely.geometry` (specifically `Polygons` and `MultiPolygons`), the `wkb` representation of the geometry, or an *Extent*.

Note: It is important that the given geometry is in the same projection as the queried layer. Otherwise, either the wrong area will be returned or an empty layer will be returned.

When the Queried Geometry is in the Same Projection as the Layer

By default, the `query` function assumes that the geometry and layer given are in the same projection.

```
layer_extent = spatial_tiled_layer.layer_metadata.extent

# Creates a Polygon from the cropped Extent of the Layer
poly = box(layer_extent.xmin+100, layer_extent.ymin+100, layer_extent.xmax-100, layer_
↪extent.ymax-100)
```

```
# Returns the region of the layer that was intersected by the Polygon at zoom level␣
↪11.
gps.query(uri="file:///tmp/spatial-catalog",
          layer_name="spatial-layer",
          layer_zoom=11,
          query_geom=poly)
```

When the Queried Geometry is in a Different Projection than the Layer

As stated above, it is important that both the geometry and layer are in the same projection. If the two are in different `CRS`s, then this can be resolved by setting the `proj_query` parameter to whatever projection the geometry is in.

```
# The queried Extent is in a different projection than the base layer
metadata = spatial_tiled_layer.tile_to_layout(layout=gps.GlobalLayout(), target_
↪crs=4326).layer_metadata
metadata.layout_definition.extent, spatial_tiled_layer.layer_metadata.layout_
↪definition.extent

# Queries the area of the Extent and returns any intersections
querried_spatial_layer = gps.query(uri="file:///tmp/spatial-catalog",
                                   layer_name="spatial-layer",
                                   layer_zoom=11,
                                   query_geom=metadata.layout_definition.extent.to_
↪polygon,
                                   query_proj="EPSG:4326")

# Because we queried the whole Extent of the layer, we should have gotten back the␣
↪whole thing.
querried_extent = querried_spatial_layer.layer_metadata.layout_definition.extent
base_extent = spatial_tiled_layer.layer_metadata.layout_definition.extent

querried_extent == base_extent
```

Querying a Spatial-Temporal Layer

In addition to being able to query a geometry, spatial-temporal data can also be filtered by time as well. These times are given as `datetime.datetime` instances.

Querying by Time

```
min_key = space_time_tiled_layer.layer_metadata.bounds.minKey

# Returns a TiledRasterLayer whose keys intersect the given time interval.
# In this case, the entire layer will be read.
gps.query(uri="file:///tmp/spacetime-catalog",
          layer_name="spacetime-layer",
          layer_zoom=7,
          time_intervals=[min_key.instant, max_key.instant])

# It's possible to query a single time interval. By doing so, only Tiles that contain␣
↪the time given will be
# returned.
gps.query(uri="file:///tmp/spacetime-catalog",
          layer_name="spacetime-layer",
          layer_zoom=7,
          time_intervals=[min_key.instant])
```

Querying by Space and Time

```
# In addition to Polygons, one can also query using MultiPolygons.
poly_1 = box(140.0, 60.0, 150.0, 65.0)
poly_2 = box(160.0, 70.0, 179.0, 89.0)
multi_poly = MultiPolygon(poly_1, poly_2)
```

(continues on next page)

(continued from previous page)

```
# Returns a TiledRasterLayer that contains the tiles which intersect the given␣
↪polygons and are within the
# specified time interval.
gps.query(uri="file:///tmp/spacetime-catalog",
          layer_name="spacetime-layer",
          layer_zoom=7,
          query_geom=multi_poly,
          time_intervals=[min_key.instant, max_key.instant])
```

Non-Intersecting Queries

In the event that neither the `query_geom` nor `time_intervals` intersects the layer, then an empty `TiledRasterLayer` will be returned.

```
# A non-intersecting geometry that we will use to query our layer.
bad_area = box(-100, -100, 0, 0)

# This will return an empty TiledRasterLayer
empty_layer = gps.query(uri="file:///tmp/spatial-catalog",
                        layer_name="spatial-layer",
                        layer_zoom=11,
                        query_geom=bad_area)

empty_layer.isEmpty()
```

2.5.8 AttributeStore

When writing a layer, GeoPySpark uses an *AttributeStore* to write layer metadata required to read and query the layer later. This class can be used outside of catalog `write` and `query` functions to inspect available layers and store additional, user defined, attributes.

Creating AttributeStore

AttributeStore can be created from the same `URI` that is given to `write` and `query` functions.

```
store = gps.AttributeStore(uri='file:///tmp/spatial-catalog')

# Check if layer exists
store.contains('spatial-layer', 11)

# List layers stored in the catalog, giving list of AttributeStore.Attributes
attributes_list = store.layers

# Ask for layer attributes by name
attributes = store.layer('spatial-layer', 11)

# Read layer metadata
attributes.layer_metadata()
```

User Defined Attributes

Internally *AttributeStore* is a key-value store where key is a tuple of layer name and zoom and values are encoded as JSON. The layer metadata is stored under attribute named `metadata`. Care should be taken to not overwrite this attribute.

```
# Reading layer metadata as underlying JSON value
attributes.read("metadata")
```

```
{'header': {'format': 'file',
  'keyClass': 'geotrellis.spark.SpatialKey',
  'path': 'spatial-layer/11',
  'valueClass': 'geotrellis.raster.MultibandTile'},
 'keyIndex': {'properties': {'keyBounds': {'maxKey': {'col': 1485, 'row': 996},
↪'minKey': {'col': 1479, 'row': 984}}},
  'type': 'zorder'},
 'metadata': {'bounds': {'maxKey': {'col': 1485, 'row': 996},
   'minKey': {'col': 1479, 'row': 984}},
  'cellType': 'int16',
  'crs': '+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0␣
↪+k=1.0 +units=m +nadgrids=@null +wktext +no_defs ',
  'extent': {'xmax': 9024345.159093022,
   'xmin': 8905559.263461886,
   'ymax': 781182.2141882492,
   'ymin': 542452.4856863784},
  'layoutDefinition': {'extent': {'xmax': 20037508.342789244,
    'xmin': -20037508.342789244,
    'ymax': 20037508.342789244,
    'ymin': -20037508.342789244},
   'tileLayout': {'layoutCols': 2048, 'layoutRows': 2048, 'tileCols': 256, 'tileRows
↪': 256}}},
 'schema': {...}
}
```

Otherwise you are free to store any additional attribute that is associated with the layer. *Attributes* provides `write` and `read` functions that accept and provide a dictionary.

```
attributes.write("notes", {'a': 3, 'b': 5})
notes_dict = attributes.read("notes")
```

A common use case for this is to store the layer histogram when writing a layer so it may be used for rendering later.

```
# Calculate the histogram
hist = spatial_tiled_layer.get_histogram()

# GeoPySpark classes have to_dict as a convention when appropriate
hist_dict = hist.to_dict()

# Writing a dictionary that gets encoded as JSON
attributes.write("histogram", hist_dict)

# Reverse the process
hist_read_dict = attributes.read("histogram")

# GeoPySpark classes have from_dict static method as a convention
hist_read = gps.Histogram.from_dict(hist_read_dict)

```

(continues on next page)

(continued from previous page)

```
# Use the histogram after round trip
hist.min_max()
```

AttributeStore Caching

An instance of *AttributeStore* keeps an in memory cache of attributes recently accessed. This is done because a common access pattern to check layer existence, read the layer and decode the layer will produce repeated requests for layer metadata. Depending on the backend used this may add considerable overhead and expense.

When writing a workflow that places heavy demand on *AttributeStore* reading it is worth while keeping track of a class instance and reusing it

```
# Retrieve already created instance if its been asked for before
store = gps.AttributeStore.cached(uri='file:///tmp/spatial-catalog-2')

# Catalog functions have optional store parameter that allows its reuse
gps.write(uri='file:///tmp/spatial-catalog-2',
          layer_name='spatial-layer',
          tiled_raster_layer=spatial_tiled_layer,
          store=store)
```

2.6 Map Algebra

Given a set of raster layers, it may be desirable to combine and filter the content of those layers. This is the function of *map algebra*. Two classes of map algebra operations are provided by GeoPySpark: *local* and *focal* operations. Local operations individually consider the pixels or cells of one or more rasters, applying a function to the corresponding cell values. For example, adding two rasters' pixel values to form a new layer is a local operation.

Focal operations consider a region around each pixel of an input raster and apply an operation to each region. The result of that operation is stored in the corresponding pixel of the output raster. For example, one might weight a 5x5 region centered at a pixel according to a 2d Gaussian to effect a blurring of the input raster. One might consider this roughly equivalent to a 2d convolution operation.

Note: Map algebra operations work only on `TiledRasterLayers`, and if a local operation requires multiple inputs, those inputs must have the same layout and projection.

Before begining, all examples in this guide need the following boilerplate code:

```
import geopyspark as gps
import numpy as np

from pyspark import SparkContext
from shapely.geometry import Point, MultiPolygon, LineString, box

conf = gps.geopyspark_conf(master="local[*]", appName="map-algebra")
pysc = SparkContext(conf=conf)

# Setting up the data

cells = np.array([[[3, 4, 1, 1, 1],
                   [7, 4, 0, 1, 0],
                   [3, 3, 7, 7, 1],
                   [0, 7, 2, 0, 0],
```

(continues on next page)

(continued from previous page)

```
                    [6, 6, 6, 5, 5]]], dtype='int32')

extent = gps.ProjectedExtent(extent = gps.Extent(0, 0, 5, 5), epsg=4326)

layer = [(extent, gps.Tile.from_numpy_array(numpy_array=cells))]

rdd = pysc.parallelize(layer)
raster_layer = gps.RasterLayer.from_numpy_rdd(gps.LayerType.SPATIAL, rdd)
tiled_layer = raster_layer.tile_to_layout(layout=gps.LocalLayout(tile_size=5))
```

2.6.1 Local Operations

Local operations on `TiledRasterLayers` can use `ints`, `floats`, or other `TiledRasterLayers`. `+`, `-`, `*`, `/`, `**`, and `abs` are all of the local operations that currently supported.

```
(tiled_layer + 1)

(2 - (tiled_layer * 3))

((tiled_layer + tiled_layer) / (tiled_layer + 1))

abs(tiled_layer)

2 ** tiled_layer
```

A *`Pyramid`* can also be used in local operations. The types that can be used in local operations with `Pyramids` are: `ints`, `floats`, `TiledRasterLayers`, and other `Pyramids`.

Note: Like with `TiledRasterLayer`, performing calculations on multiple `Pyramids` or `TiledRasterLayers` means they must all have the same layout and projection.

```
# Creating out Pyramid
pyramid = tiled_layer.pyramid()

pyramid + 1

(pyramid - tiled_layer) * 2
```

2.6.2 Focal Operations

Focal operations are performed in GeoPySpark by executing a given operation on a neighborhood throughout each tile in the layer. One can select a neighborhood to use from the `Neighborhood` enum class. Likewise, an operation can be choosen from the enum class, `Operation`.

```
# This creates an instance of Square with an extent of 1. This means that
# each operation will be performed on a 3x3
# neighborhood.

'''
A square neighborhood with an extent of 1.
o = source cell
x = cells that fall within the neighbhorhood

```

(continues on next page)

(continued from previous page)

```
x x x
x o x
x x x
'''

square = gps.Square(extent=1)
```

Mean

```
tiled_layer.focal(operation=gps.Operation.MEAN, neighborhood=square)
```

Median

```
tiled_layer.focal(operation=gps.Operation.MEDIAN, neighborhood=square)
```

Mode

```
tiled_layer.focal(operation=gps.Operation.MODE, neighborhood=square)
```

Sum

```
tiled_layer.focal(operation=gps.Operation.SUM, neighborhood=square)
```

Standard Deviation

```
tiled_layer.focal(operation=gps.Operation.STANDARD_DEVIATION, neighborhood=square)
```

Min

```
tiled_layer.focal(operation=gps.Operation.MIN, neighborhood=square)
```

Max

```
tiled_layer.focal(operation=gps.Operation.MAX, neighborhood=square)
```

Slope

```
tiled_layer.focal(operation=gps.Operation.SLOPE, neighborhood=square)
```

Aspect

```
tiled_layer.focal(operation=gps.Operation.ASPECT, neighborhood=square)
```

2.6.3 Miscellaneous Raster Operations

There are other means to extract information from rasters and to create rasters that need to be presented. These are *polygonal summaries*, *cost distance*, and *rasterization*.

Polygonal Summary Methods

In addition to local and focal operations, polygonal summaries can also be performed on `TiledRasterLayers`. These are operations that are executed in the areas that intersect a given geometry and the layer.

Note: It is important the given geometry is in the same projection as the layer. If they are not, then either incorrect and/or only partial results will be returned.

```
tiled_layer.layer_metadata
```

Polygonal Min

```
poly_min = box(0.0, 0.0, 1.0, 1.0)
tiled_layer.polygonal_min(geometry=poly_min, data_type=int)
```

Polygonal Max

```
poly_max = box(1.0, 0.0, 2.0, 2.5)
tiled_layer.polygonal_min(geometry=poly_max, data_type=int)
```

Polygonal Sum

```
poly_sum = box(0.0, 0.0, 1.0, 1.0)
tiled_layer.polygonal_min(geometry=poly_sum, data_type=int)
```

Polygonal Mean

```
poly_max = box(1.0, 0.0, 2.0, 2.0)
tiled_layer.polygonal_min(geometry=poly_max, data_type=int)
```

Cost Distance

`cost_distance()` is an iterative method for approximating the weighted distance from a raster cell to a given geometry. The `cost_distance` function takes in a geometry and a "friction layer" which essentially describes how difficult it is to traverse each raster cell. Cells that fall within the geometry have a final cost of zero, while friction cells that contain noData values will correspond to noData values in the final result. All other cells have a value that

describes the minimum cost of traversing from that cell to the geometry. If the friction layer is uniform, this function approximates the Euclidean distance, modulo some scalar value.

```
cost_distance_cells = np.array([[[1.0, 1.0, 1.0, 1.0, 1.0],
                                 [1.0, 1.0, 1.0, 1.0, 1.0],
                                 [1.0, 1.0, 1.0, 1.0, 1.0],
                                 [1.0, 1.0, 1.0, 1.0, 1.0],
                                 [1.0, 1.0, 1.0, 1.0, 0.0]]])

tile = gps.Tile.from_numpy_array(numpy_array=cost_distance_cells, no_data_value=-1.0)
cost_distance_extent = gps.ProjectedExtent(extent=gps.Extent(xmin=0.0, ymin=0.0,␣
↪xmax=5.0, ymax=5.0), epsg=4326)
cost_distance_layer = [(cost_distance_extent, tile)]

cost_distance_rdd = pysc.parallelize(cost_distance_layer)
cost_distance_raster_layer = gps.RasterLayer.from_numpy_rdd(gps.LayerType.SPATIAL,␣
↪cost_distance_rdd)
cost_distance_tiled_layer = cost_distance_raster_layer.tile_to_layout(layout=gps.
↪LocalLayout(tile_size=5))

gps.cost_distance(friction_layer=cost_distance_tiled_layer, geometries=[Point(0.0, 5.
↪0)], max_distance=144000.0)
```

Rasterization

It may be desirable to convert vector data into a raster layer. For this, we provide the *rasterize()* function, which determines the set of pixel values covered by each vector element, and assigns a supplied value to that set of pixels in a target raster. If, for example, one had a set of polygons representing counties in the US, and a value for, say, the median income within each county, a raster could be made representing these data.

GeoPySpark's `rasterize` function can take a `[shapely.geometry]`, `(shapely.geometry)`, or a `PythonRDD[shapely.geometry]`. These geometries will be converted to rasters, then tiled to a given layout, and then be returned as a `TiledRasterLayer` which contains these tiled values.

Rasterize MultiPolygons

```
raster_poly_1 = box(0.0, 0.0, 5.0, 10.0)
raster_poly_2 = box(3.0, 6.0, 15.0, 20.0)
raster_poly_3 = box(13.5, 17.0, 30.0, 20.0)

raster_multi_poly = MultiPolygon([raster_poly_1, raster_poly_2, raster_poly_3])
```

```
# Creates a TiledRasterLayer with a CRS of EPSG:4326 at zoom level 5.
gps.rasterize(geoms=[raster_multi_poly], crs=4326, zoom=5, fill_value=1)
```

Rasterize a PythonRDD of Polygons

```
poly_rdd = pysc.parallelize([raster_poly_1, raster_poly_2, raster_poly_3])

# Creates a TiledRasterLayer with a CRS of EPSG:3857 at zoom level 5.
gps.rasterize(geoms=poly_rdd, crs=3857, zoom=3, fill_value=10)
```

Rasterize LineStrings

```
line_1 = LineString(((0.0, 0.0), (0.0, 5.0)))
line_2 = LineString(((7.0, 5.0), (9.0, 12.0), (12.5, 15.0)))
line_3 = LineString(((12.0, 13.0), (14.5, 20.0)))
```

```
# Creates a TiledRasterLayer whose cells have a data type of int16.
gps.rasterize(geoms=[line_1, line_2, line_3], crs=4326, zoom=3, fill_value=2, cell_
↪type=gps.CellType.INT16)
```

Rasterize Polygons and LineStrings

```
# Creates a TiledRasterLayer from both LineStrings and MultiPolygons
gps.rasterize(geoms=[line_1, line_2, line_3, raster_multi_poly], crs=4326, zoom=5,␣
↪fill_value=2)
```

2.7 Visualizing Data in GeoPySpark

Data is visualized in GeoPySpark by running a server which allows it to be viewed in an interactive way. Before putting the data on the server, however, it must first be formatted and colored. This guide seeks to go over the steps needed to create a visualization server in GeoPySpark.

Before begining, all examples in this guide need the following boilerplate code:

```
curl -o /tmp/cropped.tif https://s3.amazonaws.com/geopyspark-test/example-files/
↪cropped.tif
```

```
import geopyspark as gps
import matplotlib.pyplot as plt

from colortools import Color
from pyspark import SparkContext

%matplotlib inline

conf = gps.geopyspark_conf(master="local[*]", appName="visualization")
pysc = SparkContext(conf=conf)

raster_layer = gps.geotiff.get(layer_type=gps.LayerType.SPATIAL, uri="/tmp/cropped.tif
↪")
tiled_layer = raster_layer.tile_to_layout(layout=gps.GlobalLayout(), target_crs=3857)
```

2.7.1 Pyramid

The *Pyramid* class represents a list of `TiledRasterLayers` that represent the same area where each layer is a level within the pyramid at a specific zoom level. Thus, as one moves up the pyramid (starting a level 0), the image will have its pixel resolution increased by a power of 2 for each level. It is this varying level of detail that allows an interactive tile server to be created from a `Pyramid`. This class is needed in order to create visualizations of the contents within its layers.

Creating a Pyramid

There are currently two different ways to create a `Pyramid` instance: Through the `TiledRasterLayer.pyramid` method or by constructing it by passing in a `[TiledRasterLayer]` or `{zoom_level: TiledRasterLayer}` to `Pyramid`.

Any `TiledRasterLayer` with a `max_zoom` can be pyramided. However, the resulting `Pyramid` may have limited functionality depending on the layout of the source `TiledRasterLayer`. In order to be used for visualization, the `Pyramid` **must** have been created from `TiledRasterLayer` that was tiled using a `GlobalLayout` and whose tiles have a spatial resolution of a power of 2.

Via the pyramid Method

When using the `Pyramid` method, a `Pyramid` instance will be created with levels from 0 to `TiledRasterlayer.zoom_level`. Thus, if a `TiledRasterLayer` has a `zoom_level` of 12 then the resulting `Pyramid` will have 13 levels that each correspond to a zoom from 0 to 12.

```
pyramided = tiled_layer.pyramid()
```

Contrusting a Pyramid Manually

```
gps.Pyramid([tiled_layer.tile_to_layout(gps.GlobalLayout(zoom=x)) for x in range(0,␣
↪13)])
```

```
gps.Pyramid({x: tiled_layer.tile_to_layout(gps.GlobalLayout(zoom=x)) for x in range(0,
↪ 13)})
```

Computing the Histogram of a Pyramid

One can produce a `Histogram` instance representing the bottom most layer within a `Pyramid` via the *`get_histogram()`* method.

```
hist = pyramided.get_histogram()
hist
```

RDD Methods

`Pyramid` contains methods for working with the `RDD`s contained within its `TiledRasterLayer`s. A list of these can be found here *RDD Methods*. When used, all internal `RDD`s will be operated on.

Map Algebra

While not as versatile as `TiledRasterLayer` in terms of map algebra operations, `Pyramid`s are still able to perform local operations between themselves, `int`s, and `float`s.

Note: Operations between two or more `Pyramid`s will occur on a per `Tile` basis which depends on the tiles having the same key. It is therefore possible to do an operation between two `Pyramid`s and getting a result where nothing has changed if neither of the `Pyramid`s have matching keys.

```
pyramided + 1

(2 * (pyramided + 2)) / 3
```

When performing operations on two or more `Pyramids`, if the `Pyamids` involved have different number of `levels`, then the resulting `Pyramid` will only have as many levels as the source `Pyramid` with the smallest level count.

```
small_pyramid = gps.Pyramid({x: tiled_layer.tile_to_layout(gps.GlobalLayout(zoom=x))␣
↪for x in range(0, 5)})
result = pyramided + small_pyramid
result.levels
```

2.7.2 ColorMap

The *`ColorMap`* class in GeoPySpark acts as a wrapper for the GeoTrellis `ColorMap` class. It is used to colorize the data within a layer when it's being visualized.

Constructing a Color Ramp

Before we can initialize `ColorMap` we must first create a list of colors (or a color ramp) to pass in. This can be created either through a function in the `color` module or manually.

Using Matplotlib

The `get_colors_from_matplotlib` function creates a color ramp using the name of on an existing in color ramp in Matplotlib and the number of colors.

Note: This function will not work if `Matplotlib` is not installed.

```
gps.get_colors_from_matplotlib(ramp_name="viridis")
```

```
gps.get_colors_from_matplotlib(ramp_name="hot", num_colors=150)
```

From ColorTools

The second helper function for constructing a color ramp is `get_colors_from_colors`. This uses the colortools package to build the ramp from `[Color]` instances.

Note: This function will not work if `colortools` is not installed.

```
colors = [Color('green'), Color('red'), Color('blue')]
colors
```

```
colors_color_ramp = gps.get_colors_from_colors(colors=colors)
colors_color_ramp
```

Creating a ColorMap

`ColorMap` has many different ways of being constructed depending on the inputs it's given.

From a Histogram

```
gps.ColorMap.from_histogram(histogram=hist, color_list=colors_color_ramp)
```

From a List of Colors

```
# Creates a ColorMap instance that will have three colors for the values that are␣
↪less than or equal to 0, 250, and
# 1000.
gps.ColorMap.from_colors(breaks=[0, 250, 1000], color_list=colors_color_ramp)
```

For NLCD Data

If the layers you are working with contain data from NLCD, then it is possible to construct a `ColorMap` without first making a color ramp and passing in a list of breaks.

```
gps.ColorMap.nlcd_colormap()
```

From a Break Map

If there aren't many colors to work with in the layer, than it may be easier to construct a `ColorMap` using a `break_map`, a `dict` that maps tile values to colors.

```
# The three tile values are 1, 2, and 3 and they correspond to the colors 0x00000000,␣
↪0x00000001, and 0x00000002
# respectively.
break_map = {
    1: 0x00000000,
    2: 0x00000001,
    3: 0x00000002
}

gps.ColorMap.from_break_map(break_map=break_map)
```

More General Build Method

As mentioned above, `ColorMap` has a more general `classmethod` called *`build()`* which takes a wide range of types to construct a `ColorMap`. In the following example, `build` will be passed the same inputs used in the previous examples.

```
# build using a Histogram
gps.ColorMap.build(breaks=hist, colors=colors_color_ramp)

# It is also possible to pass in the name of Matplotlib color ramp instead of␣
↪constructing it yourself
gps.ColorMap.build(breaks=hist, colors="viridis")

# build using Colors
gps.ColorMap.build(breaks=colors_color_ramp, colors=colors)
```

(continues on next page)

(continued from previous page)

```
# buld using breaks
gps.ColorMap.build(breaks=break_map)
```

Additional Coloring Options

In addition to supplying breaks and color values to `ColorMap`, there are other ways of changing the coloring strategy of a layer.

The following additional parameters that can be changed:

- `no_data_color`: The color of the `no_data_value` of the `Tiles`. The default is `0x00000000`
- `fallback`: The color to use when a `Tile` value has no color mapping. The default is `0x00000000`
- `classification_strategy`: How the colors should be assigned to the values based on the breaks. The default is `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO`.

2.8 TMS Servers

GeoPySpark is meant to work with geospatial data. The most natural way to interact with these data is to display them on a map. In order to allow for this interactive visualization, we provide a means to create Tile Map Service (TMS) servers directly from both GeoPySpark RDDs and tile catalogs. A TMS server may be viewed using a web-based tool such as geojson.io or interacted with using the GeoNotebook Jupyter kernel.[1]

Note that the following examples rely on this common boilerplate code:

```
import geopyspark as gps
from pyspark import SparkContext

conf = gps.geopyspark_conf(appName="demo")
sc = SparkContext(conf=conf)
```

2.8.1 Basic Example

The most straightforward use case of the TMS server is to display a singleband layer with some custom color map. This is accomplished easily:

```
cm = gps.ColorMap.nlcd_colormap()

layers = []

# Reads in the first 3 levels of the layer
for zoom in range(0, 4):
    layers.append(gps.query(uri="s3://azavea-datahub/catalog",
                            layer_name="nlcd-tms-epsg3857",
                            layer_zoom=zoom))

nlcd_pyramid = gps.Pyramid(layers)
```

(continues on next page)

[1] Note that changes allowing for display of TMS-served tiles in GeoNotebook have not yet been accepted into the master branch of that repository. In the meantime, find a TMS-enabled fork at http://github.com/geotrellis/geonotebook.

(continued from previous page)

```
tms = gps.TMS.build(source=nlcd_pyramid, display=cm)
```

Of course, other color maps can be used. See the documentation for `ColorMap` for more details.

`TMS.build` can display data from catalogs—which are represented as a string-string pair containing the URI of the catalog root and the name of the layer—or from a `Pyramid` object. One may also specify a list of any combination of these sources; more on multiple sources below.

Once a TMS server is constructed, we need to make the contents visible by binding the server. The `bind()` method can take a `host` and/or a `port`, where the former is a string, and the latter is an integer. Providing neither will result in a TMS server accessible from localhost on a random port. If the server should be accessible from the outside world, a `host` value of `"0.0.0.0"` may be used.

A call to `bind()` is then followed by a call to `url_pattern()`, which provides a string that gives the template for the tiles furnished by the TMS server. This template string may be copied directly into geojson.io, for example. When the TMS server is no longer needed, its resources can be freed by a call to `unbind()`.

```
# set up the TMS server to serve from 'localhost' on a random port
tms.bind()

tms.url_pattern

# (browse the the TMS-served layer in some interface)

tms.unbind()
```

In the event that one is using GeoPySpark from within the GeoNotebook environment, `bind` should not be used, and the following code should be used instead:

```
from geonotebook.wrappers import TMSRasterData
M.add_layer(TMSRasterData(tms), name="NLCD")
```

2.8.2 Custom Rendering Functions

For the cases when more than a simple color map needs to be applied, one may also specify a custom rendering function.[2] There are two methods for custom rendering depending on whether one is rendering a single layer or compositing multiple layers. We address each in turn.

Rendering Single Layers

If one has special demands for display—including possible ad-hoc manipulation of layer data during the display process—then one may write a Python function to convert some tile data into an image that may be served via the TMS server.

The general approach is to develop a function taking a `Tile` that returns a byte array containing the resulting image, encoded as PNG or JPG. The following example uses this rendering function approach to apply the same simple color map as above.

```
from PIL import Image
import numpy as np
```

(continues on next page)

[2] If one is only applying a colormap to a singleband tile layer, a custom rendering function should not be used as it will be noticeably slower to display.

(continued from previous page)

```
def hex_to_rgb(value):
   """Return (red, green, blue) for the color given as #rrggbb."""
   value = value.lstrip('#')
   lv = len(value)
   return tuple(int(value[i:i + lv // 3], 16) for i in range(0, lv, lv // 3))

nlcd_color_map =  { 0  : "#00000000",
                    11 : "#526095FF",     # Open Water
                    12 : "#FFFFFFFF",     # Perennial Ice/Snow
                    21 : "#D28170FF",     # Low Intensity Residential
                    22 : "#EE0006FF",     # High Intensity Residential
                    23 : "#990009FF",     # Commercial/Industrial/Transportation
                    31 : "#BFB8B1FF",     # Bare Rock/Sand/Clay
                    32 : "#969798FF",     # Quarries/Strip Mines/Gravel Pits
                    33 : "#382959FF",     # Transitional
                    41 : "#579D57FF",     # Deciduous Forest
                    42 : "#2A6B3DFF",     # Evergreen Forest
                    43 : "#A6BF7BFF",     # Mixed Forest
                    51 : "#BAA65CFF",     # Shrubland
                    61 : "#45511FFF",     # Orchards/Vineyards/Other
                    71 : "#D0CFAAFF",     # Grasslands/Herbaceous
                    81 : "#CCC82FFF",     # Pasture/Hay
                    82 : "#9D5D1DFF",     # Row Crops
                    83 : "#CD9747FF",     # Small Grains
                    84 : "#A7AB9FFF",     # Fallow
                    85 : "#E68A2AFF",     # Urban/Recreational Grasses
                    91 : "#B6D8F5FF",     # Woody Wetlands
                    92 : "#B6D8F5FF" }    # Emergent Herbaceous Wetlands

def rgba_functions(color_map):
   m = {}
   for key in color_map:
      m[key] = hex_to_rgb(color_map[key])


   def r(v):
      if v in m:
         return m[v][0]
      else:
         return 0

   def g(v):
      if v in m:
         return m[v][1]
      else:
         return 0

   def b(v):
      if v in m:
         return m[v][2]
      else:
         return 0

   def a(v):
      if v in m:
         return m[v][3]
```

(continues on next page)

(continued from previous page)

```
        else:
            return 0x00

    return (np.vectorize(r), np.vectorize(g), np.vectorize(b), np.vectorize(a))

def render_nlcd(tile):
    '''
    Assumes that the tile is a multiband tile with a single band.
    (meaning shape = (1, cols, rows))
    '''
    arr = tile.cells[0]
    (r, g, b, a) = rgba_functions(nlcd_color_map)

    rgba = np.dstack([r(arr), g(arr), b(arr), a(arr)]).astype('uint8')

    img = Image.fromarray(rgba, mode='RGBA')

    return img

tms = gps.TMS.build(nlcd_pyramid, display=render_nlcd)
```

You will likely observe noticeably slower performance compared to the earlier example. This is because the contents of each tile must be transferred from the JVM to the Python environment prior to rendering. If performance is important to you, and a color mapping solution is available, please use that approach.

Compositing Multiple Layers

It is also possible to combine data from various sources at the time of display. Of course, one could use map algebra to produce a composite layer, but if the input layers are large, this could potentially be a time-consuming operation. The TMS server allows for a list of sources to be supplied; these may be any combination of *`Pyramid`* objects and catalogs. We then may supply a function that takes a list of *`Tile`* instances and produces the bytes of an image as in the single-layer case.

The following example masks the NLCD layer to areas above 1371 meters, using some of the helper functions from the previous example.

```
from scipy.interpolate import interp2d

layers = []

for zoom in range(0, 4):
    layers.append(gps.query(uri="s3://azavea-datahub/catalog",
                            layer_name="us-ned-tms-epsg3857",
                            layer_zoom=zoom))

ned_pyramid = gps.Pyramid(layers)

def comp(tiles):
    elev256 = tiles[0].cells[0]
    grid256 = range(256)
    f = interp2d(grid256, grid256, elev256)
    grid512 = np.arange(0, 256, 0.5)
    elev = f(grid512, grid512)

    land_use = tiles[1].cells[0]
```

(continues on next page)

(continued from previous page)

```
    arr = land_use
    arr[elev < 1371] = 0

    (r, g, b, a) = rgba_functions(nlcd_color_map)

    rgba = np.dstack([r(arr), g(arr), b(arr), a(arr)]).astype('uint8')

    img = Image.fromarray(rgba, mode='RGBA')

    return img

tms = gps.TMS.build([ned_pyramid, nlcd_pyramid], display=comp)
```

This example shows the major pitfall likely to be encountered in this approach: tiles of different size must be somehow combined. NLCD tiles are 512x512, while the National Elevation Data (NED) tiles are 256x256. In this example, the NED data is (bilinearly) resampled using scipy's `interp2d` function to the proper size.

Debugging Considerations

Be aware that if there are problems in the rendering or compositing functions, the TMS server will tend to produce empty images, which can result in a silent failure of a layer to display, or odd exceptions in programs expecting meaningful images, such as GeoNotebook. It is advisable to thoroughly test these rendering functions ahead of deployment, as errors encountered in their use will be largely invisible.

2.9 Ingesting an Image

This example shows how to ingest a grayscale image and save the results locally. It is assumed that you have already read through the documentation on GeoPySpark before beginning this tutorial.

2.9.1 Getting the Data

Before we can begin with the ingest, we must first download the data from S3. This curl command will download a file from S3 and save it to your `/tmp` direcotry. The file being downloaded comes from the Shuttle Radar Topography Mission (SRTM) dataset, and contains elevation data on the east coast of Sri Lanka.

A side note: Files can be retrieved directly from S3 using the methods shown in this tutorial. However, this could not be done in this instance due to permission requirements needed to access the file.

```
curl -o /tmp/cropped.tif https://s3.amazonaws.com/geopyspark-test/example-files/
↪cropped.tif
```

2.9.2 What is an Ingest?

Before continuing on, it would be best to briefly discuss what an ingest actually is. When data is acquired, it may cover an arbitrary spatial extent in an arbitrary projection. This data needs to be regularized to some expected layout and cut into tiles. After this step, we will possess a `TiledRasterLayer` that can be analyzed and saved for later use. For more information on layers and the data they hold, see the layers guide.

2.9.3 The Code

With our file downloaded we can begin the ingest.

```
import geopyspark as gps

from pyspark import SparkContext
```

Setting Up the SparkContext

The first thing one needs to do when using GeoPySpark is to setup `SparkContext`. Because GeoPySpark is backed by Spark, the `pysc` is needed to initialize our starting classes.

For those that are already familiar with Spark, you may already know there are multiple ways to create a `SparkContext`. When working with GeoPySpark, it is advised to create this instance via `SparkConf`. There are numerous settings for `SparkConf`, and some **have** to be set a certain way in order for GeoPySpark to work. Thus, `geopyspark_conf` was created as way for a user to set the basic parameters without having to worry about setting the other, required fields.

```
conf = gps.geopyspark_conf(master="local[*]", appName="ingest-example")
pysc = SparkContext(conf=conf)
```

Reading in the Data

After the creation of `pysc`, we can now read in the data. For this example, we will be reading in a single GeoTiff that contains spatial data. Hence, why we set the `layer_type` to `LayerType.SPATIAL`.

```
raster_layer = gps.geotiff.get(layer_type=gps.LayerType.SPATIAL, uri="file:///tmp/
↪cropped.tif")
```

Tiling the Data

It is now time to format the data within the layer to our desired layout. The aptly named, `tile_to_layout`, method will cut and arrange the rasters in the layer to the layout of our choosing. This results in us getting a new class instance of `TiledRasterLayer`. For this example, we will be tiling to a `GlobalLayout`.

With our tiled data, we might like to make a tile server from it and show it in on a map at some point. Therefore, we have to make sure that the tiles within the layer are in the right projection. We can do this by setting the `target_crs` parameter.

```
tiled_raster_layer = raster_layer.tile_to_layout(gps.GlobalLayout(), target_crs=3857)
tiled_raster_layer
```

Pyramiding the Data

Now it's time to pyramid! With our reprojected data, we will create an instance of `Pyramid` that contains 12 `TiledRasterLayers`. Each one having it's own `zoom_level` from 11 to 0.

```
pyramided_layer = tiled_raster_layer.pyramid()
pyramided_layer.max_zoom
pyramided_layer.levels
```

Saving the Pyramid Locally

To save all of the `TiledRasterLayers` within `pyramid_layer`, we just have to loop through values of `pyramid_layer.level` and write each layer locally.

```
for tiled_layer in pyramided_layer.levels.values():
    gps.write(uri="file:///tmp/ingested-image", layer_name="ingested-image", tiled_
↪raster_layer=tiled_layer)
```

2.10 Reading in Sentinel-2 Images

Sentinel-2 is an observation mission developed by the European Space Agency to monitor the surface of the Earth official website. Sets of images are taken of the surface where each image corresponds to a specific wavelength. These images can provide useful data for a wide variety of industries, however, the format they are stored in can prove difficult to work with. This being `JPEG 2000` (file extension `.jp2`), an image compression format for JPEGs that allows for improved quality and compression ratio.

2.10.1 Why Use GeoPySpark

There are few libraries and/or applications that can work with `jp2`s and big data, which can make processing large amounts of sentinel data difficult. However, by using GeoPySpark in conjunction with the tools available in Python, we are able to read in and work with large sets of sentinel imagery.

2.10.2 Getting the Data

Before we can start this tutorial, we will need to get the sentinel images. All sentinel data can be found on Amazon's S3 service, and we will be downloading it straight from there.

We will download three different `jp2`s that represent the same area and time in different wavelengths: Aerosol detection (443 nm), Water vapor (945 nm), and Cirrus (1375 nm). These bands are chosen because they are all in the same 60m resolution. The tiles we will be working with cover the eastern coast of Corsica taken on January 4th, 2017.

For more information on the way the data is stored on S3, please see this link.

```
curl -o /tmp/B01.jp2 http://sentinel-s2-l1c.s3.amazonaws.com/tiles/32/T/NM/2017/1/4/0/
↪B01.jp2
curl -o /tmp/B09.jp2 http://sentinel-s2-l1c.s3.amazonaws.com/tiles/32/T/NM/2017/1/4/0/
↪B09.jp2
curl -o /tmp/B10.jp2 http://sentinel-s2-l1c.s3.amazonaws.com/tiles/32/T/NM/2017/1/4/0/
↪B10.jp2
```

2.10.3 The Code

Now that we have the files, we can begin to read them into GeoPySpark.

```
import rasterio
import geopyspark as gps
import numpy as np

from pyspark import SparkContext
```

```
conf = gps.geopyspark_conf(master="local[*]", appName="sentinel-ingest-example")
pysc = SparkContext(conf=conf)
```

2.10.4 Reading in the JPEG 2000's

`rasterio`, being backed by GDAL, allows us to read in the `jp2s`. Once they are read in, we will then combine the three seperate numpy arrays into one. This combined array represents a single, multiband raster.

```
jp2s = ["/tmp/B01.jp2", "/tmp/B09.jp2", "/tmp/B10.jp2"]
arrs = []

for jp2 in jp2s:
    with rasterio.open(jp2) as f:
        arrs.append(f.read(1))

data = np.array(arrs, dtype=arrs[0].dtype)
data
```

2.10.5 Creating the RDD

With our raster data in hand, we can how begin the creation of a Python `RDD`. Please see the core concepts guide for more information on what the following instances represent.

```
# Create an Extent instance from rasterio's bounds
extent = gps.Extent(*f.bounds)

# The EPSG code can also be obtained from the information read in via rasterio
projected_extent = gps.ProjectedExtent(extent=extent, epsg=int(f.crs.to_dict()['init
↪'][5:]))
projected_extent
```

You may have noticed in the above code that we did something weird to get the `CRS` from the rasterio file. This had to be done because the way rasterio formats the projection of the read in rasters is not compatible with how GeoPySpark expects the `CRS` to be in. Thus, we had to do a bit of extra work to get it into the correct state

```
# Projection information from the rasterio file
f.crs.to_dict()
```

```
# The projection information formatted to work with GeoPySpark
int(f.crs.to_dict()['init'][5:])
```

```
# We can create a Tile instance from our multiband, raster array and the nodata value␣
↪from rasterio
tile = gps.Tile.from_numpy_array(numpy_array=data, no_data_value=f.nodata)
tile
```

```
# Now that we have our ProjectedExtent and Tile, we can create our RDD from them
rdd = pysc.parallelize([(projected_extent, tile)])
rdd
```

2.10.6 Creating the Layer

From the `RDD`, we can now create a `RasterLayer` using the `from_numpy_rdd` method.

```
# While there is a time component to the data, this was ignored for this tutorial and␣
↪instead the focus is just
# on the spatial information. Thus, we have a LayerType of SPATIAL.
raster_layer = gps.RasterLayer.from_numpy_rdd(layer_type=gps.LayerType.SPATIAL, numpy_
↪rdd=rdd)
raster_layer
```

2.10.7 Where to Go From Here

By creating a `RasterLayer`, we can now work with and analyze the data within it. If you wish to know more about these operations, please see the following guides: Layers Guide, Map Algebra Guide Visulation Guide, and the Catalog Guide.

2.11 Reading and Rasterizing Open Street Map Data

This tutorial shows how to read in Open Street Map (OSM) data, and then rasterize it using GeoPySpark.

Note: This guide is aimed at users who are already familiar with GeoPySpark.

2.11.1 Getting the Data

To start, let's first grab an orc file, which a special file type that is optimized for Hadoop operations. The following command will use `curl` to download the file from `S3` and move it to the `/tmp` directory.

```
curl -o /tmp/boyertown.orc https://s3.amazonaws.com/geopyspark-test/example-files/
↪boyertown.orc
```

A side note: Files can be retrieved directly from S3. However, this could not be done in this instance due to permission requirements needed to access the file.

2.11.2 Reading in the Data

Now that we have our data, we can now read it in and begin to work with.

```
import geopyspark as gps
from pyspark import SparkContext

conf = gps.geopyspark_conf(appName="osm-rasterize-example", master="local[*]")
pysc = SparkContext(conf=conf)

features = gps.osm_reader.from_orc("/tmp/boyertown.orc")
```

The above code sets up a `SparkContext` and then reads in the `boyertown.orc` file as `features`, which is an instance of `FeaturesCollection`.

When OSM data is read into GeoPySpark, each OSM Element is turned into single or multiple different geometries. With each of these geometries retaining the metadata from the derived OSM Element. These geometry metadata pairs are referred to as a `Feature`. These features are grouped together by the type of geometry they contain. When accessing features from a `FeaturesCollection`, it is done by geometry.

There are four different types of geometries in the `FeaturesCollection`:

- `Point`
- `Line`
- `Polygon`
- `MultiPolygon`

Selecting the Features We Want

For this example, we're interested in rasterizing the `Lines` and `Polygons` from the OSM data, so we will select those `Features` from the `FeaturesCollection` that contain them. The following code will create a Python `RDD` of `Features` that contains all `Line` geometries (`lines`), and a Python `RDD` that contains all `Polygon` geometries (`polygons`).

```
lines = features.get_line_features_rdd()
polygons = features.get_polygon_features_rdd()
```

Looking at the Tags of the Features

When we rasterize the `Polygon Features`, we'd like for schools to have a different value than all of the other `Polygons`. However, we are unsure if any schools were contained within the original data, and we'd like to see if any are. One method we could use to determine if there are schools is to look at the tags of the `Polygon Features`. The following code will show all of the unique tags for all of the `Polygons` in the collection.

```
features.get_polygon_tags()
```

Which has the following output:

```
{'NHD:ComID': '25964412',
 'NHD:Elevation': '0.00000000000',
 'NHD:FCode': '39004',
 'NHD:FDate': '2001/08/16',
 'NHD:FTYPE': 'LakePond',
 'NHD:Permanent_': '25964412',
 'NHD:ReachCode': '02040203004486',
 'NHD:Resolution': 'High',
 'addr:city': 'Gilbertsville',
 'addr:housenumber': '1100',
 'addr:postcode': '19525',
 'addr:state': 'PA',
 'addr:street': 'E Philadelphia Avenue',
 'amenity': 'school',
 'area': 'yes',
 'building': 'yes',
 'leisure': 'pitch',
 'name': 'Boyertown Area Junior High School-West Center',
```

(continues on next page)

(continued from previous page)

```
 'natural': 'water',
 'railway': 'platform',
 'smoking': 'outside',
 'source': 'Yahoo',
 'sport': 'baseball',
 'tourism': 'museum',
 'wikidata': 'Q8069423',
 'wikipedia': "en:Zern's Farmer's Market"}
```

So it appears that there are schools in this dataset, and that we can continue on.

2.11.3 Assigning Values to Geometries

Now that we have our `Features`, it's time to assign them values. The reason we need to do so is because when a vector becomes a raster, its cells need to have some kind of value. When rasterizing `Features`, each geometry contained within it will be given a single value, and all cells that intersect that shape will have that value. In addition to value of the actual cells, there's another property that we will want to set for each `Feature`, `Z-Index`.

The `Z-Index` of a `Feature` determines what value a cell will be if more than one geometry intersects it. With a higher `Z-Index` taking priority over a lower one. This is important as there may be cases where multiple geometries are present at a single cell, but that cell can only contain one value.

For this example, we are going to want all `Polygons` to have a higher `Z-Index` than the `Lines`. In addition, since we're interested in schools, `Polygons` that are labeled as schools will have a greater `Z-Index` than other `Polygons`.

```
mapped_lines = lines.map(lambda feature: gps.Feature(feautre.geometry, gps.
↪CellValue(value=1, zindex=1)))

def assign_polygon_feature(feature):
    tags = feature.properties.tags.values()

    if 'school' in tags.values():
        return gps.Feature(feature.geometry, gps.CellValue(value=3, zindex=3))
    else:
        return gps.Feature(feature.geometry, gps.CellValue(value=2, zindex=2))

mapped_polygons = polygons.map(assign_polygon_feature)
```

We create the `mapped_lines` variable that contains an `RDD` of `Features`, where each `Feature` has a `CellValue` with a `value` and `zindex` of 1. The `assign_polygon_feature` function is then created which will test to see if a `Polygon` is a school or not. If it is, then the resulting `Feature` will have a `CellValue` with a `value` and `zindex` of 3. Otherwise, those two values will be 2.

2.11.4 Rasterizing the Features

Now that the `Features` have been given `CellValues`, it is now time to rasterize them.

```
unioned_features = pysc.union((mapped_lines, mapped_polygons))

rasterized_layer = gps.rasterize_features(features=unioned_features, crs=4326,␣
↪zoom=12)
```

The `rasterize_features` function requires a single `RDD` of `Features`. Therefore, we union together `mapped_lines` and `mapped_polygons` which gives us `unioned_features`. Along with passing in our

RDD, we must also set the `crs` and `zoom` of the resulting Layer. In this case, the `crs` is in `LatLng`, so we set it to be `4326`. `zoom` varies between use cases, so it was just chosen arbitrarily for this example. The resulting `rasterized_layer` is a `TiledRasterLayer` that we can now analyze and/or ingest.

2.12 geopyspark package

`geopyspark.`**`geopyspark_conf`**(*master=None*, *appName=None*, *additional_jar_dirs=[]*)

Construct the base SparkConf for use with GeoPySpark. This configuration object may be used as is , or may be adjusted according to the user's needs.

Note: The GEOPYSPARK_JARS_PATH environment variable may contain a colon-separated list of directories to search for JAR files to make available via the SparkConf.

Parameters

- **`master`** (*`string`*) – The master URL to connect to, such as "local" to run locally with one thread, "local[4]" to run locally with 4 cores, or "spark://master:7077" to run on a Spark standalone cluster.
- **`appName`** (*`string`*) – The name of the application, as seen in the Spark console
- **`additional_jar_dirs`** (*`list, optional`*) – A list of directory locations that might contain JAR files needed by the current script. Already includes $(pwd)/jars.

Returns SparkConf

`class` `geopyspark.`**`Tile`**

Represents a raster in GeoPySpark.

Note: All rasters in GeoPySpark are represented as having multiple bands, even if the original raster just contained one.

Parameters

- **`cells`** (*`nd.array`*) – The raster data itself. It is contained within a NumPy array.
- **`data_type`** (*`str`*) – The data type of the values within `data` if they were in Scala.
- **`no_data_value`** – The value that represents no data value in the raster. This can be represented by a variety of types depending on the value type of the raster.

`cells`

nd.array – The raster data itself. It is contained within a NumPy array.

`data_type`

str – The data type of the values within `data` if they were in Scala.

`no_data_value`

The value that represents no data value in the raster. This can be represented by a variety of types depending on the value type of the raster.

`cell_type`

Alias for field number 1

cells
Alias for field number 0

count(*value*) → integer – return number of occurrences of value

static dtype_to_cell_type(*dtype*)
Converts a `np.dtype` to the corresponding GeoPySpark `cell_type`.

Note: `bool`, `complex64`, `complex128`, and `complex256`, are currently not supported `np.dtypes`.

Parameters **dtype** (*np.dtype*) – The `dtype` of the numpy array.

Returns str. The GeoPySpark `cell_type` equivalent of the `dtype`.

Raises `TypeError` – If the given `dtype` is not a supported data type.

classmethod from_numpy_array(*numpy_array*, *no_data_value=None*)
Creates an instance of `Tile` from a numpy array.

Parameters

- **numpy_array** (*np.array*) – The numpy array to be used to represent the cell values of the `Tile`.

  Note: GeoPySpark does not support arrays with the following data types: `bool`, `complex64`, `complex128`, and `complex256`.

- **no_data_value** (*optional*) – The value that represents no data value in the raster. This can be represented by a variety of types depending on the value type of the raster. If not given, then the value will be `None`.

Returns *Tile*

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

no_data_value
Alias for field number 2

class geopyspark.**Extent**
The "bounding box" or geographic region of an area on Earth a raster represents.

Parameters

- **xmin** (*float*) – The minimum x coordinate.
- **ymin** (*float*) – The minimum y coordinate.
- **xmax** (*float*) – The maximum x coordinate.
- **ymax** (*float*) – The maximum y coordinate.

xmin
float – The minimum x coordinate.

ymin
float – The minimum y coordinate.

xmax
float – The maximum x coordinate.

ymax
float – The maximum y coordinate.

count(*value*) → integer – return number of occurrences of value

classmethod from_polygon(*polygon*)
Creates a new instance of `Extent` from a Shapely Polygon.

The new `Extent` will contain the min and max coordinates of the Polygon; regardless of the Polygon's shape.

Parameters **polygon** (*shapely.geometry.Polygon*) – A Shapely Polygon.

Returns *Extent*

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

to_polygon
Converts this instance to a Shapely Polygon.

The resulting Polygon will be in the shape of a box.

Returns `shapely.geometry.Polygon`

xmax
Alias for field number 2

xmin
Alias for field number 0

ymax
Alias for field number 3

ymin
Alias for field number 1

class geopyspark.**ProjectedExtent**
Describes both the area on Earth a raster represents in addition to its CRS.

Parameters

- **extent** (*Extent*) – The area the raster represents.
- **epsg** (*int, optional*) – The EPSG code of the CRS.
- **proj4** (*str, optional*) – The Proj.4 string representation of the CRS.

extent
Extent – The area the raster represents.

epsg
int, optional – The EPSG code of the CRS.

proj4
str, optional – The Proj.4 string representation of the CRS.

Note: Either `epsg` or `proj4` must be defined.

count(*value*) → integer – return number of occurrences of value

epsg
Alias for field number 1

extent
Alias for field number 0

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

proj4
Alias for field number 2

class geopyspark.**TemporalProjectedExtent**
Describes the area on Earth the raster represents, its CRS, and the time the data was collected.

Parameters

- **extent** (*Extent*) – The area the raster represents.
- **instant** (datetime.datetime) – The time stamp of the raster.
- **epsg** (*int, optional*) – The EPSG code of the CRS.
- **proj4** (*str, optional*) – The Proj.4 string representation of the CRS.

extent
Extent – The area the raster represents.

instant
datetime.datetime – The time stamp of the raster.

epsg
int, optional – The EPSG code of the CRS.

proj4
str, optional – The Proj.4 string representation of the CRS.

Note: Either epsg or proj4 must be defined.

count (*value*) → integer – return number of occurrences of value

epsg
Alias for field number 2

extent
Alias for field number 0

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

instant
Alias for field number 1

proj4
Alias for field number 3

class geopyspark.**SpatialKey**
Represents the position of a raster within a grid. This grid is a 2D plane where raster positions are represented by a pair of coordinates.

Parameters

- **col** (*int*) – The column of the grid, the numbers run east to west.

- **row** (*int*) – The row of the grid, the numbers run north to south.

col
int – The column of the grid, the numbers run east to west.

row
int – The row of the grid, the numbers run north to south.

col
Alias for field number 0

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

row
Alias for field number 1

class geopyspark.**SpaceTimeKey**
Represents the position of a raster within a grid. This grid is a 3D plane where raster positions are represented by a pair of coordinates as well as a z value that represents time.

Parameters

- **col** (*int*) – The column of the grid, the numbers run east to west.
- **row** (*int*) – The row of the grid, the numbers run north to south.
- **instant** (datetime.datetime) – The time stamp of the raster.

col
int – The column of the grid, the numbers run east to west.

row
int – The row of the grid, the numbers run north to south.

instant
datetime.datetime – The time stamp of the raster.

col
Alias for field number 0

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

instant
Alias for field number 2

row
Alias for field number 1

class geopyspark.**Metadata** (*bounds*, *crs*, *cell_type*, *extent*, *layout_definition*)
Information of the values within a RasterLayer or TiledRasterLayer. This data pertains to the layout and other attributes of the data within the classes.

Parameters

- **bounds** (*Bounds*) – The Bounds of the values in the class.
- **crs** (*str or int*) – The CRS of the data. Can either be the EPSG code, well-known name, or a PROJ.4 projection string.
- **cell_type** (str or *CellType*) – The data type of the cells of the rasters.

- **extent** (*Extent*) – The Extent that covers the all of the rasters.
- **layout_definition** (*LayoutDefinition*) – The LayoutDefinition of all rasters.

bounds
Bounds – The Bounds of the values in the class.

crs
str or int – The CRS of the data. Can either be the EPSG code, well-known name, or a PROJ.4 projection string.

cell_type
str – The data type of the cells of the rasters.

no_data_value
int or float or None – The noData value of the rasters within the layer. This can either be None, an int, or a float depending on the cell_type.

extent
Extent – The Extent that covers the all of the rasters.

tile_layout
TileLayout – The TileLayout that describes how the rasters are orginized.

layout_definition
LayoutDefinition – The LayoutDefinition of all rasters.

classmethod from_dict(*metadata_dict*)
Creates Metadata from a dictionary.

Parameters metadata_dict (*dict*) – The Metadata of a RasterLayer or TiledRasterLayer instance that is in dict form.

Returns *Metadata*

to_dict()
Converts this instance to a dict.

Returns dict

class geopyspark.**TileLayout**
Describes the grid in which the rasters within a Layer should be laid out.

Parameters

- **layoutCols** (*int*) – The number of columns of rasters that runs east to west.
- **layoutRows** (*int*) – The number of rows of rasters that runs north to south.
- **tileCols** (*int*) – The number of columns of pixels in each raster that runs east to west.
- **tileRows** (*int*) – The number of rows of pixels in each raster that runs north to south.

layoutCols
int – The number of columns of rasters that runs east to west.

layoutRows
int – The number of rows of rasters that runs north to south.

tileCols
int – The number of columns of pixels in each raster that runs east to west.

tileRows
int – The number of rows of pixels in each raster that runs north to south.

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

layoutCols
Alias for field number 0

layoutRows
Alias for field number 1

tileCols
Alias for field number 2

tileRows
Alias for field number 3

class geopyspark.**GlobalLayout**
TileLayout type that spans global CRS extent.

When passed in place of LayoutDefinition it signifies that a LayoutDefinition instance should be constructed such that it fits the global CRS extent. The cell resolution of resulting layout will be one of resolutions implied by power of 2 pyramid for that CRS. Tiling to this layout will likely result in either up-sampling or down-sampling the source raster.

Parameters

- **tile_size** (*int*) – The number of columns and row pixels in each tile.
- **zoom** (*int, optional*) – Override the zoom level in power of 2 pyramid.
- **threshold** (*float, optional*) – The percentage difference between a cell size and a zoom level and the resolution difference between that zoom level and the next that is tolerated to snap to the lower-resolution zoom level. For example, if this paramter is 0.1, that means we're willing to downsample rasters with a higher resolution in order to fit them to some zoom level Z, if the difference is resolution is less than or equal to 10% the difference between the resolutions of zoom level Z and zoom level Z+1.

tile_size
int – The number of columns and row pixels in each tile.

zoom
int – The desired zoom level of the layout.

threshold
float, optional – The percentage difference between a cell size and a zoom level and the resolution difference between that zoom level and the next that is tolerated to snap to the lower-resolution zoom level.

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

threshold
Alias for field number 2

tile_size
Alias for field number 0

zoom
Alias for field number 1

class geopyspark.**LocalLayout**
TileLayout type that snaps the layer extent.

When passed in place of LayoutDefinition it signifies that a LayoutDefinition instances should be constructed over the envelope of the layer pixels with given tile size. Resulting TileLayout will match the cell resolution of the source rasters.

Parameters

- **tile_size** (*int, optional*) – The number of columns and row pixels in each tile. If this is `None`, then the sizes of each tile will be set using `tile_cols` and `tile_rows`.
- **tile_cols** (*int, optional*) – The number of column pixels in each tile. This supersedes `tile_size`. Meaning if this and `tile_size` are set, then this will be used for the number of colunn pixles. If `None`, then the number of column pixels will default to 256.
- **tile_rows** (*int, optional*) – The number of rows pixels in each tile. This supersedes `tile_size`. Meaning if this and `tile_size` are set, then this will be used for the number of row pixles. If `None`, then the number of row pixels will default to 256.

tile_cols
int – The number of column pixels in each tile

tile_rows
int – The number of rows pixels in each tile. This supersedes

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

tile_cols
Alias for field number 0

tile_rows
Alias for field number 1

class geopyspark.**LayoutDefinition**
Describes the layout of the rasters within a Layer and how they are projected.

Parameters

- **extent** (*Extent*) – The `Extent` of the layout.
- **tileLayout** (*TileLayout*) – The `TileLayout` of how the rasters within the Layer.

extent
Extent – The `Extent` of the layout.

tileLayout
TileLayout – The `TileLayout` of how the rasters within the Layer.

count (*value*) → integer – return number of occurrences of value

extent
Alias for field number 0

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

tileLayout
Alias for field number 1

class geopyspark.**Bounds**
Represents the grid that covers the area of the rasters in a Layer on a grid.

Parameters

- **minKey** (*SpatialKey* or *SpaceTimeKey*) – The smallest `SpatialKey` or `SpaceTimeKey`.
- **minKey** – The largest `SpatialKey` or `SpaceTimeKey`.

minKey
SpatialKey or *SpaceTimeKey* – The smallest `SpatialKey` or `SpaceTimeKey`.

minKey
SpatialKey or *SpaceTimeKey* – The largest `SpatialKey` or `SpaceTimeKey`.

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

maxKey
Alias for field number 1

minKey
Alias for field number 0

class geopyspark.**RasterizerOptions**
Represents options available to geometry rasterizer

Parameters

- **includePartial** (*bool, optional*) – Include partial pixel intersection (default: True)
- **sampleType** (*str, optional*) – 'PixelIsArea' or 'PixelIsPoint' (default: 'PixelIs-Point')

includePartial
bool – Include partial pixel intersection.

sampleType
str – How the sampling should be performed during rasterization.

count(*value*) → integer – return number of occurrences of value

includePartial
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

sampleType
Alias for field number 1

geopyspark.**zfactor_lat_lng_calculator**(*unit*)
Produces the Scala class, `ZFactorCalculator` as a `JavaObject`.

The resulting `ZFactorCalculator` produced using this method assumes that the `Tiles` it will be deriving `zfactors` from are in `LatLng` (aka `epsg:4326`). This caculator can still be used on `Tiles` with different projections, however, the resulting `Slope` calculations may be off.

Parameters **units** (str or `Unit`) – The unit of elevation in the target layer.

Returns `py4j.JavaObject`

geopyspark.**zfactor_calculator**(*mapped_zfactors*)
Produces the Scala class, `ZFactorCalculator` as a `JavaObject`.

Unlike the `ZFactorCalculator` produced in `zfactor_lat_lng_calculator()`, this resulting `ZFactorCalculator` can used on `Tiles` in a different projection. However, it cannot be used between different types of projections. For example, a `ZFactorCalculator` produced for a Layer that is in `WebMercator` will not create an accurate `ZFactor` for a Layer that is in `LatLng`.

Parameters **`mapped_zfactors`** (*`dict`*) – A `dict` that maps lattitudes to `ZFactors`. It is not required to supply a mapping for ever lattitude intersected in the layer. Rather, based on the lattitudes given, a linear interpolation will be performed and any lattitude not mapped will have its `ZFactor` derived from that interpolation.

Returns `py4j.JavaObject`

class `geopyspark.`**`HashPartitionStrategy`**

Represents a partitioning strategy for a layer that uses Spark's `HashPartitioner` with a set number of partitions.

Parameters **`num_partitions`** (*`int, optional`*) – The number of partitions that should be used during partitioning. Default is, `None`. If `None` the resulting layer will have a `HashPartitioner` with the number of partitions being either the same as the input layer's, or a number computed by the method.

`count`(*value*) → integer – return number of occurrences of value

`index`(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`num_partitions`
Alias for field number 0

class `geopyspark.`**`SpatialPartitionStrategy`**

Represents a partitioning strategy for a layer that uses GeoPySpark's `SpatialPartitioner` with a set number of partitions.

This partitioner will try and group `Tiles` together that are spatially near each other in the same partition. In order to do this, each `Tile` has their `Key Index` calculated using the space filling curve index, `Z-Curve`.

Parameters

- **`num_partitions`** (*`int, optional`*) – The number of partitions that should be used during partitioning. Default is, `None`. If `None` the resulting layer will have a `HashPartitioner` with the number of partitions being either the same as the input layer's, or a number computed by the method.
- **`bits`** (*`int, optional`*) – Helps determine how much data should be placed in each partition. Default is, `8`.

  GeoPySpark uses a Z-order curve to determine how values within the layer should be grouped. This is done by first finding the `Key Index` of a value and then performing a bitwise right shift on the resulting index. From the remaining bits, a partition is selected such that those indexes with the same remaining bits will be in the same partition. Therefore, as the number of bits shifted to the right increases, so then too does the group sizes.

`num_partitions`
int – The number of partitions that should be used during partitioning.

`bits`
int – Determine how much data should be placed in each partition.

`bits`
Alias for field number 1

`count`(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

num_partitions
Alias for field number 0

class geopyspark.**SpaceTimePartitionStrategy**
Represents a partitioning strategy for a layer that uses GeoPySpark's `SpaceTimePartitioner` with a set number of partitions, units of time, and temporal resolution.

This partitioner will try and group `Tiles` together that are spatially and temproally near each other in the same partition. In order to do this, each `Tile` has their `Key Index` calculated using the space filling curve index, `Z-Curve`.

Note: This partitiong strategy will only work on `SPACETIME` layers, and will fail if given a `SPATIAL` one. For `SPATIAL` layers, please see `SpatialPartitionStrategy`.

Parameters

- **time_unit** (str or *TimeUnit*) – Which time unit should be used when saving spatial-temporal data. This controls the resolution of each index. Meaning, what time intervals are used to seperate each record.
- **num_partitions** (*int, optional*) – The number of partitions that should be used during partitioning. Default is, `None`. If `None` the resulting layer will have a `HashPartitioner` with the number of partitions being either the same as the input layer's, or a number computed by the method.
- **bits** (*int, optional*) – Helps determine how much data should be placed in each partition. Default is, `8`.

  GeoPySpark uses a Z-order curve to determine how values within the layer should be grouped. This is done by first finding the `Key Index` of a value and then performing a bitwise right shift on the resulting index. From the remaining bits, a partition is selected such that those indexes with the same remaining bits will be in the same partition. Therefore, as the number of bits shifted to the right increases, so then too does the group sizes.
- **time_resolution** (*str or int, optional*) – Determines how data for each `time_unit` should be grouped together. By default, no grouping will occur.

  As an example, having a `time_unit` of `WEEKS` and a `time_resolution` of 5 will cause the data to be grouped and stored together in units of 5 weeks. If however `time_resolution` is not specified, then the data will be grouped and stored in units of single weeks.

  This value can either be an `int` or a string representation of an `int`.

time_unit
str or *TimeUnit* – Which time unit should be used when saving spatial-temporal data.

num_partitions
int – The number of partitions that should be used during partitioning.

bits
int – Helps determine how much data should be placed in each partition.

time_resolution
str or int – Determines how data for each `time_unit` should be grouped together.

bits
Alias for field number 2

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

num_partitions
Alias for field number 1

time_resolution
Alias for field number 3

time_unit
Alias for field number 0

geopyspark.**read_layer_metadata**(*uri*, *layer_name*, *layer_zoom*)
Reads the metadata from a saved layer without reading in the whole layer.

Parameters

- **uri** (*str*) – The Uniform Resource Identifier used to point towards the desired GeoTrellis catalog to be read from. The shape of this string varies depending on backend.
- **layer_name** (*str*) – The name of the GeoTrellis catalog to be read from.
- **layer_zoom** (*int*) – The zoom level of the layer that is to be read.

Returns *Metadata*

geopyspark.**read_value**(*uri*, *layer_name*, *layer_zoom*, *col*, *row*, *zdt=None*)
Reads a single `Tile` from a GeoTrellis catalog. Unlike other functions in this module, this will not return a `TiledRasterLayer`, but rather a GeoPySpark formatted raster.

Note: When requesting a tile that does not exist, `None` will be returned.

Parameters

- **uri** (*str*) – The Uniform Resource Identifier used to point towards the desired GeoTrellis catalog to be read from. The shape of this string varies depending on backend.
- **layer_name** (*str*) – The name of the GeoTrellis catalog to be read from.
- **layer_zoom** (*int*) – The zoom level of the layer that is to be read.
- **col** (*int*) – The col number of the tile within the layout. Cols run east to west.
- **row** (*int*) – The row number of the tile within the layout. Row run north to south.
- **zdt** (`datetime.datetime`) – The time stamp of the tile if the data is spatial-temporal. This is represented as a `datetime.datetime.` instance. The default value is, `None`. If `None`, then only the spatial area will be queried.

Returns *Tile*

geopyspark.**query**(*uri*, *layer_name*, *layer_zoom=None*, *query_geom=None*, *time_intervals=None*, *query_proj=None*, *num_partitions=None*)
Queries a single, zoom layer from a GeoTrellis catalog given spatial and/or time parameters.

Note: The whole layer could still be read in if `intersects` and/or `time_intervals` have not been set, or if the querried region contains the entire layer.

Parameters

- **`layer_type`** (str or *`LayerType`*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **`uri`** (*`str`*) – The Uniform Resource Identifier used to point towards the desired GeoTrellis catalog to be read from. The shape of this string varies depending on backend.
- **`layer_name`** (*`str`*) – The name of the GeoTrellis catalog to be querried.
- **`layer_zoom`** (*`int, optional`*) – The zoom level of the layer that is to be querried. If `None`, then the `layer_zoom` will be set to 0.
- **`query_geom`** (bytes or shapely.geometry or *`Extent`*, Optional) – The desired spatial area to be returned. Can either be a string, a shapely geometry, or instance of `Extent`, or a WKB verson of the geometry.

  Note: Not all shapely geometires are supported. The following is are the types that are supported: * Point * Polygon * MultiPolygon

  Note: Only layers that were made from spatial, singleband GeoTiffs can query a `Point`. All other types are restricted to `Polygon` and `MulitPolygon`.

  Note: If the queried region does not intersect the layer, then an empty layer will be returned.

  If not specified, then the entire layer will be read.
- **`time_intervals`** (`[datetime.datetime]`, optional) – A list of the time intervals to query. This parameter is only used when querying spatial-temporal data. The default value is, `None`. If `None`, then only the spatial area will be querried.
- **`query_proj`** (*`int or str, optional`*) – The crs of the querried geometry if it is different than the layer it is being filtered against. If they are different and this is not set, then the returned `TiledRasterLayer` could contain incorrect values. If `None`, then the geometry and layer are assumed to be in the same projection.
- **`num_partitions`** (*`int, optional`*) – Sets RDD partition count when reading from catalog.

Returns *`TiledRasterLayer`*

`geopyspark.`**`write`**(*uri*, *layer_name*, *tiled_raster_layer*, *index_strategy=<IndexingMethod.ZORDER: 'zorder'>*, *time_unit=None*, *time_resolution=None*, *store=None*)

Writes a tile layer to a specified destination.

Parameters

- **`uri`** (*`str`*) – The Uniform Resource Identifier used to point towards the desired location for the tile layer to written to. The shape of this string varies depending on backend.
- **`layer_name`** (*`str`*) – The name of the new, tile layer.

- **layer_zoom** (*int*) – The zoom level the layer should be saved at.
- **tiled_raster_layer** (*TiledRasterLayer*) – The `TiledRasterLayer` to be saved.
- **index_strategy** (str or *IndexingMethod*) – The method used to orginize the saved data. Depending on the type of data within the layer, only certain methods are available. Can either be a string or a `IndexingMethod` attribute. The default method used is, `IndexingMethod.ZORDER`.
- **time_unit** (str or *TimeUnit*, optional) – Which time unit should be used when saving spatial-temporal data. This controls the resolution of each index. Meaning, what time intervals are used to seperate each record. While this is set to `None` as default, it must be set if saving spatial-temporal data. Depending on the indexing method chosen, different time units are used.
- **time_resolution** (*str or int, optional*) – Determines how data for each `time_unit` should be grouped together. By default, no grouping will occur.

  As an example, having a `time_unit` of `WEEKS` and a `time_resolution` of 5 will cause the data to be grouped and stored together in units of 5 weeks. If however `time_resolution` is not specified, then the data will be grouped and stored in units of single weeks.

  This value can either be an `int` or a string representation of an `int`.
- **store** (str or *AttributeStore*, optional) – `AttributeStore` instance or URI for layer metadata lookup.

class geopyspark.**AttributeStore**(*uri*)

AttributeStore provides a way to read and write GeoTrellis layer attributes.

Internally all attribute values are stored as JSON, here they are exposed as dictionaries. Classes often stored have a `.from_dict` and `.to_dict` methods to bridge the gap:

```
import geopyspark as gps
store = gps.AttributeStore("s3://azavea-datahub/catalog")
hist = store.layer("us-nlcd2011-30m-epsg3857", zoom=7).read("histogram")
hist = gps.Histogram.from_dict(hist)
```

class Attributes(*store*, *layer_name*, *layer_zoom*)

Accessor class for all attributes for a given layer

delete(*name*)

Delete attribute by name

Parameters name (*str*) – Attribute name

layer_metadata()

read(*name*)

Read layer attribute by name as a dict

Parameters name (*str*) –

Returns Attribute value

Return type `dict`

write(*name*, *value*)

Write layer attribute value as a dict

Parameters
- **name** (*str*) – Attribute name
- **value** (*dict*) – Attribute value

classmethod build(*store*)
Builds AttributeStore from URI or passes an instance through.

Parameters **uri** (`str or AttributeStore`) – URI for AttributeStore object or instance.

Returns *AttributeStore*

classmethod cached(*uri*)
Returns cached version of AttributeStore for URI or creates one

contains(*name*, *zoom=None*)
Checks if this store contains a layer metadata.

Parameters

- **name** (*str*) – Layer name
- **zoom** (*int, optional*) – Layer zoom

Returns `bool`

delete(*name*, *zoom=None*)
Delete layer and all its attributes

Parameters

- **name** (*str*) – Layer name
- **zoom** (*int, optional*) – Layer zoom

layer(*name*, *zoom=None*)
Layer Attributes object for given layer :param name: Layer name :type name: str :param zoom: Layer zoom :type zoom: int, optional

Returns `Attributes`

layers()
List all layers Attributes objects

Returns `[:class:`~geopyspark.geotrellis.catalog.AttributeStore.Attributes`]`

geopyspark.**get_colors_from_colors**(*colors*)
Returns a list of integer colors from a list of Color objects from the colortools package.

Parameters **colors** (*[colortools.Color]*) – A list of color stops using colortools.Color

Returns [int]

geopyspark.**get_colors_from_matplotlib**(*ramp_name*, *num_colors=256*)
Returns a list of color breaks from the color ramps defined by Matplotlib.

Parameters

- **ramp_name** (*str*) – The name of a matplotlib color ramp. See the matplotlib documentation for a list of names and details on each color ramp.
- **num_colors** (*int, optional*) – The number of color breaks to derive from the named map.

Returns [int]

class geopyspark.**ColorMap**(*cmap*)
A class that wraps a GeoTrellis ColorMap class.

Parameters **cmap** (*py4j.java_gateway.JavaObject*) – The `JavaObject` that represents the GeoTrellis ColorMap.

cmap
py4j.java_gateway.JavaObject – The `JavaObject` that represents the GeoTrellis ColorMap.

classmethod build(*breaks*, *colors=None*, *no_data_color=0*, *fallback=0*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)
Given breaks and colors, build a `ColorMap` object.

Parameters

- **breaks** (dict or list or `np.ndarray` or `Histogram`) – If a `dict` then a mapping from tile values to colors, the latter represented as integers e.g., 0xff000080 is red at half opacity. If a `list` then tile values that specify breaks in the color mapping. If a `Histogram` then a histogram from which breaks can be derived.
- **colors** (*str or list, optional*) – If a `str` then the name of a matplotlib color ramp. If a `list` then either a list of colortools `Color` objects or a list of integers containing packed RGBA values. If `None`, then the `ColorMap` will be created from the `breaks` given.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

classmethod from_break_map(*break_map*, *no_data_color=0*, *fallback=0*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)
Converts a dictionary mapping from tile values to colors to a ColorMap.

Parameters

- **break_map** (*dict*) – A mapping from tile values to colors, the latter represented as integers e.g., 0xff000080 is red at half opacity.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

classmethod from_colors(*breaks*, *color_list*, *no_data_color=0*, *fallback=0*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)
Converts lists of values and colors to a `ColorMap`.

Parameters

- **breaks** (*list*) – The tile values that specify breaks in the color mapping.
- **color_list** (*[int]*) – The colors corresponding to the values in the breaks list, represented as integers—e.g., 0xff000080 is red at half opacity.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

classmethod from_histogram(*histogram, color_list, no_data_color=0, fallback=0, classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)

Converts a wrapped GeoTrellis histogram into a `ColorMap`.

Parameters

- **histogram** (`Histogram`) – A `Histogram` instance; specifies breaks
- **color_list** (*[int]*) – The colors corresponding to the values in the breaks list, represented as integers e.g., 0xff000080 is red at half opacity.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

static nlcd_colormap()

Returns a color map for NLCD tiles.

Returns *ColorMap*

class geopyspark.**LayerType**

The type of the key within the tuple of the wrapped RDD.

SPACETIME = 'spacetime'

SPATIAL = 'spatial'

Indicates that the RDD contains `(K, V)` pairs, where the `K` has a spatial and time attribute. Both *TemporalProjectedExtent* and *SpaceTimeKey* are examples of this type of `K`.

class geopyspark.**IndexingMethod**

How the wrapped should be indexed when saved.

HILBERT = 'hilbert'
A key indexing method. Works only for RDDs that contain *SpatialKey*. This method provides the fastest lookup of all the key indexing method, however, it does not give good locality guarantees. It is recommended then that this method should only be used when locality is not important for your analysis.

ROWMAJOR = 'rowmajor'

ZORDER = 'zorder'
A key indexing method. Works for RDDs that contain both *SpatialKey* and *SpaceTimeKey*. Note, indexes are determined by the `x`, `y`, and if `SPACETIME`, the temporal resolutions of a point. This is expressed in bits, and has a max value of 62. Thus if the sum of those resolutions are greater than 62, then the indexing will fail.

class geopyspark.**ResampleMethod**
Resampling Methods.

AVERAGE = 'Average'

BILINEAR = 'Bilinear'

CUBIC_CONVOLUTION = 'CubicConvolution'

CUBIC_SPLINE = 'CubicSpline'

LANCZOS = 'Lanczos'

MAX = 'Max'

MEDIAN = 'Median'

MIN = 'Min'

MODE = 'Mode'

NEAREST_NEIGHBOR = 'NearestNeighbor'

class geopyspark.**TimeUnit**
ZORDER time units.

DAYS = 'days'

HOURS = 'hours'

MILLIS = 'millis'

MINUTES = 'minutes'

MONTHS = 'months'

SECONDS = 'seconds'

WEEKS = 'weeks'

YEARS = 'years'

class geopyspark.**Operation**
Focal opertions.

ASPECT = 'Aspect'

MAX = 'Max'

MEAN = 'Mean'

MEDIAN = 'Median'

MIN = 'Min'

MODE = 'Mode'

STANDARD_DEVIATION = 'StandardDeviation'

SUM = 'Sum'

VARIANCE = 'Variance'

class geopyspark.**Neighborhood**

Neighborhood types.

ANNULUS = 'Annulus'

CIRCLE = 'Circle'

NESW = 'Nesw'

SQUARE = 'Square'

WEDGE = 'Wedge'

class geopyspark.**ClassificationStrategy**

Classification strategies for color mapping.

EXACT = 'Exact'

GREATER_THAN = 'GreaterThan'

GREATER_THAN_OR_EQUAL_TO = 'GreaterThanOrEqualTo'

LESS_THAN = 'LessThan'

LESS_THAN_OR_EQUAL_TO = 'LessThanOrEqualTo'

class geopyspark.**CellType**

Cell types.

BOOL = 'bool'

BOOLRAW = 'boolraw'

FLOAT32 = 'float32'

FLOAT32RAW = 'float32raw'

FLOAT64 = 'float64'

FLOAT64RAW = 'float64raw'

INT16 = 'int16'

INT16RAW = 'int16raw'

INT32 = 'int32'

INT32RAW = 'int32raw'

INT8 = 'int8'

INT8RAW = 'int8raw'

UINT16 = 'uint16'

UINT16RAW = 'uint16raw'

UINT8 = 'uint8'

UINT8RAW = 'uint8raw'

class geopyspark.**ColorRamp**

ColorRamp names.

BLUE_TO_ORANGE = 'BlueToOrange'

BLUE_TO_RED = 'BlueToRed'

CLASSIFICATION_BOLD_LAND_USE = 'ClassificationBoldLandUse'

CLASSIFICATION_MUTED_TERRAIN = 'ClassificationMutedTerrain'

COOLWARM = 'CoolWarm'

GREEN_TO_RED_ORANGE = 'GreenToRedOrange'

HEATMAP_BLUE_TO_YELLOW_TO_RED_SPECTRUM = 'HeatmapBlueToYellowToRedSpectrum'

HEATMAP_DARK_RED_TO_YELLOW_WHITE = 'HeatmapDarkRedToYellowWhite'

HEATMAP_LIGHT_PURPLE_TO_DARK_PURPLE_TO_WHITE = 'HeatmapLightPurpleToDarkPurpleToWhite'

HEATMAP_YELLOW_TO_RED = 'HeatmapYellowToRed'

Hot = 'Hot'

INFERNO = 'Inferno'

LIGHT_TO_DARK_GREEN = 'LightToDarkGreen'

LIGHT_TO_DARK_SUNSET = 'LightToDarkSunset'

LIGHT_YELLOW_TO_ORANGE = 'LightYellowToOrange'

MAGMA = 'Magma'

PLASMA = 'Plasma'

VIRIDIS = 'Viridis'

class geopyspark.**StorageMethod**

Internal storage methods for GeoTiffs.

STRIPED = 'Striped'

TILED = 'Tiled'

class geopyspark.**ColorSpace**

Color space types for GeoTiffs.

BLACK_IS_ZERO = 1

CFA = 32803

CIE_LAB = 8

CMYK = 5

ICC_LAB = 9

ITU_LAB = 10

LINEAR_RAW = 34892

LOG_L = 32844

LOG_LUV = 32845

PALETTE = 3

RGB = 2

TRANSPARENCY_MASK = 4

WHITE_IS_ZERO = 0

Y_CB_CR = 6

class `geopyspark.`**Compression**

Compression methods for GeoTiffs.

DEFLATE_COMPRESSION = 'DeflateCompression'

NO_COMPRESSION = 'NoCompression'

class `geopyspark.`**Unit**

Represents the units of elevation.

FEET = 'Feet'

METERS = 'Meters'

`geopyspark.`**cost_distance**(*friction_layer*, *geometries*, *max_distance*)

Performs cost distance of a TileLayer.

Parameters

- **friction_layer** (*TiledRasterLayer*) – `TiledRasterLayer` of a friction surface to traverse.
- **geometries** (*list*) – A list of shapely geometries to be used as a starting point.

  Note: All geometries must be in the same CRS as the TileLayer.

- **max_distance** (*int or float*) – The maximum cost that a path may reach before the operation. stops. This value can be an `int` or `float`.

Returns *TiledRasterLayer*

`geopyspark.`**euclidean_distance**(*geometry*, *source_crs*, *zoom*, *cell_type=<CellType.FLOAT64: 'float64'>*)

Calculates the Euclidean distance of a Shapely geometry.

Parameters

- **geometry** (*shapely.geometry*) – The input geometry to compute the Euclidean distance for.
- **source_crs** (*str or int*) – The CRS of the input geometry.
- **zoom** (*int*) – The zoom level of the output raster.
- **cell_type** (str or *CellType*, optional) – The data type of the cells for the new layer. If not specified, then `CellType.FLOAT64` is used.

Note: This function may run very slowly for polygonal inputs if they cover many cells of the output raster.

Returns `TiledRasterLayer`

`geopyspark.`**hillshade**(*tiled_raster_layer*, *zfactor_calculator*, *band=0*, *azimuth=315.0*, *altitude=45.0*)

Computes Hillshade (shaded relief) from a raster.

The resulting raster will be a shaded relief map (a hill shading) based on the sun altitude, azimuth, and the `zfactor`. The `zfactor` is a conversion factor from map units to elevation units.

The `hillshade`` operation will be carried out in a `SQUARE` neighborhood with with an `extent` of 1. The `zfactor` will be derived from the `zfactor_calculator` for each `Tile` in the Layer. The resulting Layer will have a `cell_type` of `INT16` regardless of the input Layer's `cell_type`; as well as have a single band, that represents the calculated `hillshade`.

Returns a raster of ShortConstantNoDataCellType.

For descriptions of parameters, please see Esri Desktop's description of Hillshade.

Parameters

- **tiled_raster_layer** (*TiledRasterLayer*) – The base layer that contains the rasters used to compute the hillshade.
- **zfactor_calculator** (*py4j.JavaObject*) – A `JavaObject` that represents the Scala `ZFactorCalculator` class. This can be created using either the `zfactor_lat_lng_calculator()` or the `zfactor_calculator()` methods.
- **band** (*int, optional*) – The band of the raster to base the hillshade calculation on. Default is 0.
- **azimuth** (*float, optional*) – The azimuth angle of the source of light. Default value is 315.0.
- **altitude** (*float, optional*) – The angle of the altitude of the light above the horizon. Default is 45.0.

Returns *TiledRasterLayer*

class geopyspark.**Histogram**(*scala_histogram*)

A wrapper class for a GeoTrellis Histogram.

The underlying histogram is produced from the values within a *TiledRasterLayer*. These values represented by the histogram can either be `Int` or `Float` depending on the data type of the cells in the layer.

Parameters **scala_histogram** (*py4j.JavaObject*) – An instance of the GeoTrellis histogram.

scala_histogram

py4j.JavaObject – An instance of the GeoTrellis histogram.

bin_counts()

Returns a list of tuples where the key is the bin label value and the value is the label's respective count.

Returns [(int, int)] or [(float, int)]

bucket_count()

Returns the number of buckets within the histogram.

Returns int

cdf()

Returns the cdf of the distribution of the histogram.

Returns [(float, float)]

classmethod from_dict(*value*)

Encodes histogram as a dictionary

item_count(*item*)

Returns the total number of times a given item appears in the histogram.

Parameters **item** (*int or float*) – The value whose occurences should be counted.

Returns The total count of the occurences of `item` in the histogram.

Return type int

max()
The largest value of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns int or float

mean()
Determines the mean of the histogram.

Returns float

median()
Determines the median of the histogram.

Returns float

merge(*other_histogram*)
Merges this instance of `Histogram` with another. The resulting `Histogram` will contain values from both ``Histogram``s

Parameters **other_histogram** (*Histogram*) – The `Histogram` that should be merged with this instance.

Returns *Histogram*

min()
The smallest value of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns int or float

min_max()
The largest and smallest values of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns (int, int) or (float, float)

mode()
Determines the mode of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns int or float

quantile_breaks(*num_breaks*)
Returns quantile breaks for this Layer.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns [int]

to_dict()
Encodes histogram as a dictionary

Returns `dict`

values()
Lists each indiviual value within the histogram.

This will return a list of either ``int``s or ``float``s depedning on the type of values within the histogram.

Returns [int] or [float]

class `geopyspark.`**`RasterLayer`**(*layer_type*, *srdd*)

A wrapper of a RDD that contains GeoTrellis rasters.

Represents a layer that wraps a RDD that contains `(K, V)`. Where `K` is either *`ProjectedExtent`* or *`TemporalProjectedExtent`* depending on the `layer_type` of the RDD, and `V` being a *`Tile`*.

The data held within this layer has not been tiled. Meaning the data has yet to be modified to fit a certain layout. See raster_rdd for more information.

Parameters

- **`layer_type`** (str or *`LayerType`*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **`srdd`** (*`py4j.java_gateway.JavaObject`*) – The coresponding Scala class. This is what allows `RasterLayer` to access the various Scala methods.

`pysc`

pyspark.SparkContext – The `SparkContext` being used this session.

`layer_type`

`LayerType` – What the layer type of the geotiffs are.

`srdd`

py4j.java_gateway.JavaObject – The coresponding Scala class. This is what allows `RasterLayer` to access the various Scala methods.

`bands`(*band*)

Select a subsection of bands from the `Tiles` within the layer.

Note: There could be potential high performance cost if operations are performed between two sub-bands of a large data set.

Note: Due to the natue of GeoPySpark's backend, if selecting a band that is out of bounds then the error returned will be a `py4j.protocol.Py4JJavaError` and not a normal Python error.

Parameters **`band`** (*`int or tuple or list or range`*) – The band(s) to be selected from the `Tiles`. Can either be a single int, or a collection of ints.

Returns *`RasterLayer`* with the selected bands.

`cache`()

Persist this RDD with the default storage level (C{MEMORY_ONLY}).

`collect_keys`()

Returns a list of all of the keys in the layer.

Note: This method should only be called on layers with a smaller number of keys, as a large number could cause memory issues.

Returns `` [:class:`~geopyspark.geotrellis.SpatialKey`] `` or `` [:ob:`~geopyspark.geotrellis.SpaceTimeKey`] ``

`collect_metadata`(*layout=LocalLayout(tile_cols=256, tile_rows=256)*)

Iterate over the RDD records and generates layer metadata desribing the contained rasters.

:param layout (`LayoutDefinition` or: `GlobalLayout` or

`LocalLayout`, optional): Target raster layout for the tiling operation.

Returns `Metadata`

`convert_data_type`(*new_type*, *no_data_value=None*)
Converts the underlying, raster values to a new `CellType`.

Parameters

- **`new_type`** (str or `CellType`) – The data type the cells should be to converted to.
- **`no_data_value`** (*`int or float, optional`*) – The value that should be marked as NoData.

Returns `RasterLayer`

Raises

- `ValueError` – If `no_data_value` is set and the `new_type` contains raw values.
- `ValueError` – If `no_data_value` is set and `new_type` is a boolean.

`count`()
Returns how many elements are within the wrapped RDD.

Returns The number of elements in the RDD.

Return type Int

`filter_by_times`(*time_intervals*)
Filters a `SPACETIME` layer by keeping only the values whose keys fall within a the given time interval(s).

Parameters **`time_intervals`** (`[datetime.datetime]`) – A list of the time intervals to query. This list can have one or multiple elements. If just a single element, then only exact matches with that given time will be kept. If there are multiple times given, then they are each paired together so that they form ranges of time. In the case where there are an odd number of elements, then the remaining time will be treated as a single query and not a range.

Note: If nothing intersects the given `time_intervals`, then the returned `RasterLayer` will be empty.

Returns `RasterLayer`

`classmethod from_numpy_rdd`(*layer_type*, *numpy_rdd*)
Create a `RasterLayer` from a numpy RDD.

Parameters

- **`layer_type`** (str or `LayerType`) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **`numpy_rdd`** (*`pyspark.RDD`*) – A PySpark RDD that contains tuples of either `ProjectedExtent`s or `TemporalProjectedExtent`s and rasters that are represented by a numpy array.

Returns `RasterLayer`

`getNumPartitions`()
Returns the number of partitions set for the wrapped RDD.

Returns The number of partitions.

Return type Int

get_class_histogram()
Creates a Histogram of integer values. Suitable for classification rasters with limited number values. If only single band is present histogram is returned directly.

Returns *Histogram* or [*Histogram*]

get_histogram()
Creates a Histogram for each band in the layer. If only single band is present histogram is returned directly.

Returns *Histogram* or [*Histogram*]

get_min_max()
Returns the maximum and minimum values of all of the rasters in the layer.

Returns (float, float)

get_partition_strategy()
Returns the partitioning strategy if the layer has one.

Returns HashPartitioner or SpatialPartitioner or *SpaceTimePartitionStrategy* or None

get_quantile_breaks(*num_breaks*)
Returns quantile breaks for this Layer.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns [float]

get_quantile_breaks_exact_int(*num_breaks*)
Returns quantile breaks for this Layer. This version uses the FastMapHistogram, which counts exact integer values. If your layer has too many values, this can cause memory errors.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns [int]

isEmpty()
Returns a bool that is True if the layer is empty and False if it is not.

Returns Are there elements within the layer

Return type bool

layer_type

map_cells(*func*)
Maps over the cells of each Tile within the layer with a given function.

Note: This operation first needs to deserialize the wrapped RDD into Python and then serialize the RDD back into a TiledRasterRDD once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **func** (*cells, nd => cells*) – A function that takes two arguements: cells and nd. Where cells is the numpy array and nd is the no_data_value of the Tile. It returns cells which are the new cells values of the Tile represented as a numpy array.

Returns *RasterLayer*

map_tiles(*func*)
Maps over each `Tile` within the layer with a given function.

Note: This operation first needs to deserialize the wrapped `RDD` into Python and then serialize the `RDD` back into a `RasterRDD` once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **func** (*Tile* => *Tile*) – A function that takes a `Tile` and returns a `Tile`.

Returns *RasterLayer*

merge(*partition_strategy=None*)
Merges the `Tile` of each `K` together to produce a single `Tile`.

This method will reduce each value by its key within the layer to produce a single `(K, V)` for every `K`. In order to achieve this, each `Tile` that shares a `K` is merged together to form a single `Tile`. This is done by replacing one `Tile`'s cells with another's. Not all cells, if any, may be replaced, however. The following steps are taken to determine if a cell's value should be replaced:

1. If the cell contains a `NoData` value, then it will be replaced.
2. If no `NoData` value is set, then a cell with a value of 0 will be replaced.
3. If neither of the above are true, then the cell retain its value.

Parameters

- **num_partitions** (*int, optional*) – The number of partitions that the resulting layer should be partitioned with. If `None`, then the `num_partitions` will the number of partitions the layer curretly has.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *RasterLayer*

partitionBy(*partition_strategy=None*)
Repartitions the layer using the given partitioning strategy.

Parameters **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

If `None`, then the output layer will be the same as the source layer.

If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns `RasterLayer`

`persist`(*storageLevel=StorageLevel(False, True, False, False, 1)*)
Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (C{MEMORY_ONLY}).

`pysc`

`reclassify`(*value_map, data_type, classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>, replace_nodata_with=None, fallback_value=None, strict=False*)
Changes the cell values of a raster based on how the data is broken up in the given `value_map`.

Parameters

- **`value_map`** (*`dict`*) – A `dict` whose keys represent values where a break should occur and its values are the new value the cells within the break should become.
- **`data_type`** (*`type`*) – The type of the values within the rasters. Can either be int or float.
- **`classification_strategy`** (str or *`ClassificationStrategy`*, optional) – How the cells should be classified along the breaks. If unspecified, then `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` will be used.
- **`replace_nodata_with`** (*`int or float, optional`*) – When remapping values, `NoData` values must be treated separately. If `NoData` values are intended to be replaced during the reclassify, this variable should be set to the intended value. If unspecified, `NoData` values will be preserved.

  Note: Specifying `replace_nodata_with` will change the value of given cells, but the `NoData` value of the layer will remain unchanged.

- **`fallback_value`** (*`int or float, optional`*) – Represents the value that should be used when a cell's value does not fall within the `classification_strategy`. Default is to use the layer's `NoData` value.
- **`strict`** (*`bool, optional`*) – Determines whether or not an error should be thrown if a cell's value does not fall within the `classification_strategy`. Default is, `False`.

Returns `RasterLayer`

`repartition`(*num_partitions=None*)
Repartitions the layer to have a different number of partitions.

Parameters **`num_partitions`** (*`int, optional`*) – Desired number of partitions. Default is, `None` .If `None`, then the exisiting number of partitions will be used.

Returns `RasterLayer`

`reproject`(*target_crs, resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*)
Reproject rasters to `target_crs`. The reproject does not sample past tile boundary.

Parameters

- **target_crs** (*str or int*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string.
- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use for the reprojection. If none is specified, then `ResampleMethods.NEAREST_NEIGHBOR` is used.

Returns *RasterLayer*

srdd

tile_to_layout(*layout=LocalLayout(tile_cols=256, tile_rows=256), target_crs=None, resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>, partition_strategy=None*)

Cut tiles to layout and merge overlapping tiles. This will produce unique keys.

Parameters

- **layout** (*Metadata* or `TiledRasterLayer` or *LayoutDefinition* or *GlobalLayout* or *LocalLayout*) – Target raster layout for the tiling operation.
- **target_crs** (*str or int, optional*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string. If `None`, no reproject will be perfomed.
- **resample_method** (str or *ResampleMethod*, optional) – The cell resample method to used during the tiling operation. Default is``ResampleMethods.NEAREST_NEIGHBOR``.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

to_geotiff_rdd(*storage_method=<StorageMethod.STRIPED: 'Striped'>, rows_per_strip=None, tile_dimensions=(256, 256), compression=<Compression.NO_COMPRESSION: 'NoCompression'>, color_space=<ColorSpace.BLACK_IS_ZERO: 1>, color_map=None, head_tags=None, band_tags=None*)

Converts the rasters within this layer to GeoTiffs which are then converted to bytes. This is returned as a `RDD[(K, bytes)]`. Where `K` is either `ProjectedExtent` or `TemporalProjectedExtent`.

Parameters

- **storage_method** (str or *StorageMethod*, optional) – How the segments within the GeoTiffs should be arranged. Default is `StorageMethod.STRIPED`.
- **rows_per_strip** (*int, optional*) – How many rows should be in each strip segment of the GeoTiffs if `storage_method` is `StorageMethod.STRIPED`. If `None`, then the strip size will default to a value that is 8K or less.

- **tile_dimensions** (*(int, int), optional*) – The length and width for each tile segment of the GeoTiff if `storage_method` is `StorageMethod.TILED`. If `None` then the default size is `(256, 256)`.
- **compression** (str or *Compression*, optional) – How the data should be compressed. Defaults to `Compression.NO_COMPRESSION`.
- **color_space** (str or *ColorSpace*, optional) – How the colors should be organized in the GeoTiffs. Defaults to `ColorSpace.BLACK_IS_ZERO`.
- **color_map** (*ColorMap*, optional) – A `ColorMap` instance used to color the GeoTiffs to a different gradient.
- **head_tags** (*dict, optional*) – A `dict` where each key and value is a `str`.
- **band_tags** (*list, optional*) – A `list` of `dict`s where each key and value is a `str`.
- **Note** – For more information on the contents of the tags, see www.gdal.org/gdal_datamodel.html

Returns RDD[(K, bytes)]

to_numpy_rdd()

Converts a `RasterLayer` to a numpy RDD.

Note: Depending on the size of the data stored within the RDD, this can be an exspensive operation and should be used with caution.

Returns RDD

to_png_rdd(*color_map*)

Converts the rasters within this layer to PNGs which are then converted to bytes. This is returned as a RDD[(K, bytes)].

Parameters **color_map** (*ColorMap*) – A `ColorMap` instance used to color the PNGs.

Returns RDD[(K, bytes)]

to_spatial_layer(*target_time=None*)

Converts a `RasterLayer` with a `layout_type` of `LayoutType.SPACETIME` to a `RasterLayer` with a `layout_type` of `LayoutType.SPATIAL`.

Parameters **target_time** (`datetime.datetime`, optional) – The instance of interest. If set, the resulting `RasterLayer` will only contain keys that contained the given instance. If `None`, then all values within the layer will be kept.

Returns *RasterLayer*

Raises `ValueError` – If the layer already has a `layout_type` of `LayoutType.SPATIAL`.

unpersist()

Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

with_no_data(*no_data_value*)

Changes the `NoData` value of the layer with the new given value.

It is possible to specify a `NoData` value for layers with raw values. The resulting layer will be of the same `CellType` but with a user defined `NoData` value. For example, if a layer has a `CellType` of `float32raw` and a `no_data_value` of `-10` is given, then the produced layer will have a `CellType` of `float32ud-10.0`.

If the target layer has a `bool` `CellType`, then the `no_data_value` will be ignored and the result layer will be the same as the origin. In order to assign a `NoData` value to a `bool` layer, the *`convert_data_type()`* method must be used.

Parameters **`no_data_value`** (*`int or float`*) – The new `NoData` value of the layer.

Returns *`RasterLayer`*

`wrapped_rdds`()

Returns the list of RDD-containing objects wrapped by this object. The default implementation assumes that subclass contains a single RDD container, srdd, which implements the persist() and unpersist() methods.

class geopyspark.**TiledRasterLayer**(*layer_type*, *srdd*)

Wraps a RDD of tiled, GeoTrellis rasters.

Represents a RDD that contains `(K, V)`. Where `K` is either *`SpatialKey`* or *`SpaceTimeKey`* depending on the `layer_type` of the RDD, and `V` being a *`Tile`*.

The data held within the layer is tiled. This means that the rasters have been modified to fit a larger layout. For more information, see tiled-raster-rdd.

Parameters

- **`layer_type`** (str or *`LayerType`*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **`srdd`** (*`py4j.java_gateway.JavaObject`*) – The coresponding Scala class. This is what allows `TiledRasterLayer` to access the various Scala methods.

`pysc`

pyspark.SparkContext – The `SparkContext` being used this session.

`layer_type`

`LayerType` – What the layer type of the geotiffs are.

`srdd`

py4j.java_gateway.JavaObject – The coresponding Scala class. This is what allows `RasterLayer` to access the various Scala methods.

`is_floating_point_layer`

bool – Whether the data within the `TiledRasterLayer` is floating point or not.

`layer_metadata`

`Metadata` – The layer metadata associated with this layer.

`zoom_level`

int – The zoom level of the layer. Can be `None`.

`aggregate_by_cell`(*operation*)

Computes an aggregate summary for each cell of all of the values for each key.

The `operation` given is a local map algebra function that will be applied to all values that share the same key. If there are multiple copies of the same key in the layer, then this method will reduce all instances of the `(K, Tile)` pairs into a single element. This resulting `(K, Tile)`'s `Tile` will contain the aggregate summaries of each cell of the reduced `Tiles` that had the same `K`.

Note: Not all `Operations` are supported. Only `SUM`, `MIN`, `MAX`, `MEAN`, `VARIANCE`, AND `STANDARD_DEVIATION` can be used.

Note: If calculating `VARIANCE` or `STANDARD_DEVIATION`, then any `K` that is a single copy will have a resulting `Tile` that is filled with `NoData` values. This is because the variance of a single element is undefined.

Parameters **`operation`** (str or *`Operation`*) – The aggregate operation to be performed.

Returns *`TiledRasterLayer`*

`bands`(*band*)

Select a subsection of bands from the `Tiles` within the layer.

Note: There could be potential high performance cost if operations are performed between two sub-bands of a large data set.

Note: Due to the natue of GeoPySpark's backend, if selecting a band that is out of bounds then the error returned will be a `py4j.protocol.Py4JJavaError` and not a normal Python error.

Parameters **`band`** (*`int or tuple or list or range`*) – The band(s) to be selected from the `Tiles`. Can either be a single int, or a collection of ints.

Returns *`TiledRasterLayer`* with the selected bands.

`cache`()

Persist this RDD with the default storage level (C{MEMORY_ONLY}).

`collect_keys`()

Returns a list of all of the keys in the layer.

Note: This method should only be called on layers with a smaller number of keys, as a large number could cause memory issues.

Returns `[:class:`~geopyspark.geotrellis.ProjectedExtent`]` or `[:class:`~geopyspark.geotrellis.TemporalProjectedExtent`]`

`convert_data_type`(*new_type*, *no_data_value=None*)

Converts the underlying, raster values to a new `CellType`.

Parameters

- **`new_type`** (str or *`CellType`*) – The data type the cells should be to converted to.
- **`no_data_value`** (*`int or float, optional`*) – The value that should be marked as NoData.

Returns *`TiledRasterLayer`*

Raises

- `ValueError` – If `no_data_value` is set and the `new_type` contains raw values.
- `ValueError` – If `no_data_value` is set and `new_type` is a boolean.

count()

Returns how many elements are within the wrapped RDD.

Returns The number of elements in the RDD.

Return type Int

filter_by_times(*time_intervals*)

Filters a `SPACETIME` layer by keeping only the values whose keys fall within a the given time interval(s).

Parameters **time_intervals** (`[datetime.datetime]`) – A list of the time intervals to query. This list can have one or multiple elements. If just a single element, then only exact matches with that given time will be kept. If there are multiple times given, then they are each paired together so that they form ranges of time. In the case where there are an odd number of elements, then the remaining time will be treated as a single query and not a range.

Note: If nothing intersects the given `time_intervals`, then the returned `TiledRasterLayer` will be empty.

Returns *TiledRasterLayer*

focal(*operation*, *neighborhood=None*, *param_1=None*, *param_2=None*, *param_3=None*, *partition_strategy=None*)

Performs the given focal operation on the layers contained in the Layer.

Parameters

- **operation** (str or *Operation*) – The focal operation to be performed.
- **neighborhood** (str or `Neighborhood`, optional) – The type of neighborhood to use in the focal operation. This can be represented by either an instance of `Neighborhood`, or by a constant.
- **param_1** (*int or float, optional*) – The first argument of `neighborhood`.
- **param_2** (*int or float, optional*) – The second argument of the `neighborhood`.
- **param_3** (*int or float, optional*) – The third argument of the `neighborhood`.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Note: `param` only need to be set if `neighborhood` is not an instance of `Neighborhood` or if `neighborhood` is `None`.

Any `param` that is not set will default to 0.0.

If `neighborhood` is `None` then `operation` **must** be `Operation.ASPECT`.

Returns *`TiledRasterLayer`*

Raises

- `ValueError` – If `operation` is not a known operation.
- `ValueError` – If `neighborhood` is not a known neighborhood.
- `ValueError` – If `neighborhood` was not set, and `operation` is not `Operation.ASPECT`.

classmethod from_numpy_rdd(*layer_type*, *numpy_rdd*, *metadata*, *zoom_level=None*)

Create a `TiledRasterLayer` from a numpy RDD.

Parameters

- **layer_type** (str or *`LayerType`*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **numpy_rdd** (*`pyspark.RDD`*) – A PySpark RDD that contains tuples of either *`SpatialKey`* or *`SpaceTimeKey`* and rasters that are represented by a numpy array.
- **metadata** (*`Metadata`*) – The `Metadata` of the `TiledRasterLayer` instance.
- **zoom_level** (*`int, optional`*) – The `zoom_level` the resulting *TiledRasterLayer* should have. If `None`, then the returned layer's `zoom_level` will be `None`.

Returns *`TiledRasterLayer`*

getNumPartitions()

Returns the number of partitions set for the wrapped RDD.

Returns The number of partitions.

Return type Int

get_class_histogram()

Creates a `Histogram` of integer values. Suitable for classification rasters with limited number values. If only single band is present histogram is returned directly.

Returns *`Histogram`* or [*`Histogram`*]

get_histogram()

Creates a `Histogram` for each band in the layer. If only single band is present histogram is returned directly.

Returns *`Histogram`* or [*`Histogram`*]

get_min_max()

Returns the maximum and minimum values of all of the rasters in the layer.

Returns (float, float)

get_partition_strategy()

Returns the partitioning strategy if the layer has one.

Returns `HashPartitioner` or `SpatialPartitioner` or *`SpaceTimePartitionStrategy`* or `None`

`get_point_values`(*points*, *resample_method=None*)

Returns the values of the layer at given points.

Note: Only points that are contained within a layer will be sampled. This means that if a point lies on the southern or eastern boundary of a cell, it will not be sampled.

Parameters

- **`or {k`** (*`points([shapely.geometry.Point])`* – shapely.geometry.Point}): Either a list of, or a dictionary whose values are `shapely.geometry.Points`. If a dictionary, then the type of its keys does not matter. These points must be in the same projection as the tiles within the layer.
- **`resample_method`** (str or `ResampleMethod`, optional) – The resampling method to use before obtaining the point values. If not specified, then `None` is used.

  Note: Not all `ResampleMethods` can be used to resample point values. `ResampleMethod.NEAREST_NEIGHBOR`, `ResampleMethod.BILINEAR``, `ResampleMethod.CUBIC_CONVOLUTION`, and `ResampleMethod.CUBIC_SPLINE` are the only ones that can be used.

Returns

The return type will vary depending on the type of `points` and the `layer_type` of the sampled layer.

If `points` is a `list` and the `layer_type` is `SPATIAL`: [(shapely.geometry.Point, [float])]

If `points` is a `list` and the `layer_type` is `SPACETIME`: [(shapely.geometry.Point, [(datetime.datetime, [float])])]

If `points` is a `dict` and the `layer_type` is `SPATIAL`: {k: (shapely.geometry.Point, [float])}

If `points` is a `dict` and the `layer_type` is `SPACETIME`: {k: (shapely.geometry.Point, [(datetime.datetime, [float])])}

The `shapely.geometry.Point` in all of these returns is the original sampled point given. The `[float]` are the sampled values, one for each band. If the `layer_type` was `SPACETIME`, then the timestamp will also be included in the results represented by a `datetime.datetime` instance. These times and their associated values will be given as a list of tuples for each point.

Note: The sampled values will always be returned as `floats`. Regardless of the `cellType` of the layer.

If `points` was given as a `dict` then the keys of that dictionary will be the keys in the returned `dict`.

`get_quantile_breaks`(*num_breaks*)

Returns quantile breaks for this Layer.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns [float]

get_quantile_breaks_exact_int (*num_breaks*)

Returns quantile breaks for this Layer. This version uses the FastMapHistogram, which counts exact integer values. If your layer has too many values, this can cause memory errors.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns [int]

histogram_series (*geometries*)

isEmpty ()

Returns a bool that is True if the layer is empty and False if it is not.

Returns Are there elements within the layer

Return type bool

layer_type

local_max (*value*)

Determines the maximum value for each cell of each Tile in the layer.

This method takes a max_constant that is compared to each cell in the layer. If max_constant is larger, then the resulting cell value will be that value. Otherwise, that cell will retain its original value.

Note: NoData values are handled such that taking the max between a normal value and NoData value will always result in NoData.

Parameters **value** (int or float or *TiledRasterLayer*) – The constant value that will be compared to each cell. If this is a TiledRasterLayer, then Tiles who share a key will have each of their cell values compared.

Returns *TiledRasterLayer*

lookup (*col*, *row*)

Return the value(s) in the image of a particular SpatialKey (given by col and row).

Parameters

- **col** (*int*) – The SpatialKey column.
- **row** (*int*) – The SpatialKey row.

Returns [*Tile*]

Raises

- ValueError – If using lookup on a non LayerType.SPATIAL TiledRasterLayer.
- IndexError – If col and row are not within the TiledRasterLayer's bounds.

map_cells (*func*)

Maps over the cells of each Tile within the layer with a given function.

Note: This operation first needs to deserialize the wrapped RDD into Python and then serialize the RDD back into a TiledRasterRDD once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **func** (*cells, nd => cells*) – A function that takes two arguements: `cells` and `nd`. Where `cells` is the numpy array and `nd` is the `no_data_value` of the tile. It returns `cells` which are the new cells values of the tile represented as a numpy array.

Returns *TiledRasterLayer*

map_tiles (*func*)
Maps over each `Tile` within the layer with a given function.

Note: This operation first needs to deserialize the wrapped `RDD` into Python and then serialize the `RDD` back into a `TiledRasterRDD` once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **func** (*Tile* => *Tile*) – A function that takes a `Tile` and returns a `Tile`.

Returns *TiledRasterLayer*

mask (*geometries*, *partition_strategy=None*, *options=RasterizerOptions(includePartial=True*, *sampleType='PixelIsPoint')*)
Masks the `TiledRasterLayer` so that only values that intersect the geometries will be available.

Parameters

- **geometries** (*shapely.geometry or [shapely.geometry] or pyspark.RDD[shapely.geometry]*) – Either a single, list, or Python `RDD` of shapely geometry/ies to mask the layer.

  Note: All geometries must be in the same CRS as the TileLayer.

- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

  Note: This parameter will only be used if `geometries` is a `pyspark.RDD`.

- **options** (*RasterizerOptions*, optional) – During the mask operation, rasterization occurs. These options will change the pixel rasterization behavior. Default behavior is to include partial pixel intersection and to treat pixels as points.

Note: This parameter will only be used if `geometries` is a `pyspark.RDD`.

Returns *`TiledRasterLayer`*

`max_series`(*geometries*)

`mean_series`(*geometries*)

`merge`(*partition_strategy=None*)

Merges the `Tile` of each `K` together to produce a single `Tile`.

This method will reduce each value by its key within the layer to produce a single `(K, V)` for every `K`. In order to achieve this, each `Tile` that shares a `K` is merged together to form a single `Tile`. This is done by replacing one `Tile`'s cells with another's. Not all cells, if any, may be replaced, however. The following steps are taken to determine if a cell's value should be replaced:

1. If the cell contains a `NoData` value, then it will be replaced.
2. If no `NoData` value is set, then a cell with a value of 0 will be replaced.
3. If neither of the above are true, then the cell retain its value.

Parameters

- **`num_partitions`** (*`int, optional`*) – The number of partitions that the resulting layer should be partitioned with. If `None`, then the `num_partitions` will the number of partitions the layer curretly has.
- **`partition_strategy`** (*`HashPartitionStrategy`* or `SpatialPartitioinStrategy` or *`SpaceTimePartitionStrategy`*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *`TiledRasterLayer`*

`min_series`(*geometries*)

`normalize`(*new_min, new_max, old_min=None, old_max=None*)

Finds the min value that is contained within the given geometry.

Note: If `old_max - old_min <= 0` or `new_max - new_min <= 0`, then the normalization will fail.

Parameters

- **`old_min`** (*`int or float, optional`*) – Old minimum. If not given, then the minimum value of this layer will be used.

- **old_max** (*int or float, optional*) – Old maximum. If not given, then the minimum value of this layer will be used.
- **new_min** (*int or float*) – New minimum to normalize to.
- **new_max** (*int or float*) – New maximum to normalize to.

Returns *TiledRasterLayer*

partitionBy(*partition_strategy=None*)
Repartitions the layer using the given partitioning strategy.

Parameters **partition_strategy** (*HashPartitionStrategy* or SpatialPartitioinStrategy or *SpaceTimePartitionStrategy*, optional) – Sets the Partitioner for the resulting layer and how many partitions it has. Default is, None.

If None, then the output layer will be the same as the source layer.

If partition_strategy is set but has no num_partitions, then the resulting layer will have the Partioner specified in the strategy with the with same number of partitions the source layer had.

If partition_strategy is set and has a num_partitions, then the resulting layer will have the Partioner and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

persist(*storageLevel=StorageLevel(False, True, False, False, 1)*)
Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (C{MEMORY_ONLY}).

polygonal_max(*geometry, data_type*)
Finds the max value for each band that is contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely Polygon or MultiPolygon that represents the area where the summary should be computed; or a WKB representation of the geometry.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.

Returns [int] or [float] depending on data_type.

Raises TypeError – If data_type is not an int or float.

polygonal_mean(*geometry*)
Finds the mean of all of the values for each band that are contained within the given geometry.

Parameters **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely Polygon or MultiPolygon that represents the area where the summary should be computed; or a WKB representation of the geometry.

Returns [float]

polygonal_min(*geometry, data_type*)
Finds the min value for each band that is contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKB representation of the geometry.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.

Returns [int] or [float] depending on `data_type`.

Raises `TypeError` – If `data_type` is not an int or float.

polygonal_sum (*geometry*, *data_type*)

Finds the sum of all of the values in each band that are contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKB representation of the geometry.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.

Returns [int] or [float] depending on `data_type`.

Raises `TypeError` – If `data_type` is not an int or float.

pyramid (*resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*, *partition_strategy=None*)

Creates a layer `Pyramid` where the resolution is halved per level.

Parameters

- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use when building the pyramid. Default is `ResampleMethods.NEAREST_NEIGHBOR`.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *Pyramid*.

Raises `ValueError` – If this layer layout is not of `GlobalLayout` type.

pysc

reclassify (*value_map*, *data_type*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*, *replace_nodata_with=None*, *fallback_value=None*, *strict=False*)

Changes the cell values of a raster based on how the data is broken up in the given `value_map`.

Parameters

- **value_map** (*dict*) – A `dict` whose keys represent values where a break should occur and its values are the new value the cells within the break should become.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.
- **classification_strategy** (str or *ClassificationStrategy*, optional) – How the cells should be classified along the breaks. If unspecified, then `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` will be used.
- **replace_nodata_with** (*int or float, optional*) – When remapping values, `NoData` values must be treated separately. If `NoData` values are intended to be replaced during the reclassify, this variable should be set to the intended value. If unspecified, `NoData` values will be preserved.

Note: Specifying `replace_nodata_with` will change the value of given cells, but the `NoData` value of the layer will remain unchanged.

- **fallback_value** (*int or float, optional*) – Represents the value that should be used when a cell's value does not fall within the `classification_strategy`. Default is to use the layer's `NoData` value.
- **strict** (*bool, optional*) – Determines whether or not an error should be thrown if a cell's value does not fall within the `classification_strategy`. Default is, `False`.

Returns *TiledRasterLayer*

repartition(*num_partitions=None*)

Repartitions the layer to have a different number of partitions.

Parameters **num_partitions** (*int, optional*) – Desired number of partitions. Default is, `None` .If `None`, then the exisiting number of partitions will be used.

Returns *TiledRasterLayer*

reproject(*target_crs*, *resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*)

Reproject rasters to `target_crs`. The reproject does not sample past tile boundary.

Parameters

- **target_crs** (*str or int*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string.
- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use for the reprojection. If none is specified, then `ResampleMethods.NEAREST_NEIGHBOR` is used.

Returns *TiledRasterLayer*

save_stitched(*path*, *crop_bounds=None*, *crop_dimensions=None*)

Stitch all of the rasters within the Layer into one raster and then saves it to a given path.

Parameters

- **path** (*str*) – The path of the geotiff to save. The path must be on the local file system.
- **crop_bounds** (*Extent*, optional) – The sub `Extent` with which to crop the raster before saving. If `None`, then the whole raster will be saved.

- **crop_dimensions** (*tuple(int) or list(int), optional*) – cols and rows of the image to save represented as either a tuple or list. If None then all cols and rows of the raster will be save.

Note: This can only be used on LayerType.SPATIAL TiledRasterLayers.

Note: If crop_dimensions is set then crop_bounds must also be set.

slope (*zfactor_calculator*)
Performs the Slope, focal operation on the first band of each Tile in the Layer.

The Slope operation will be carried out in a SQUARE neighborhood with with an extent of 1. A zfactor will be derived from the zfactor_calculator for each Tile in the Layer. The resulting Layer will have a cell_type of FLOAT64 regardless of the input Layer's cell_type; as well as have a single band, that represents the calculated slope.

Parameters **zfactor_calculator** (*py4j.JavaObject*) – A JavaObject that represents the Scala ZFactorCalculator class. This can be created using either the zfactor_lat_lng_calculator() or the zfactor_calculator() methods.

Returns *TiledRasterLayer*

srdd

star_series (*geometries*, *fn*)

stitch ()
Stitch all of the rasters within the Layer into one raster.

Note: This can only be used on LayerType.SPATIAL TiledRasterLayers.

Returns *Tile*

sum_series (*geometries*)

tile_to_layout (*layout*, *target_crs=None*, *resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*, *partition_strategy=None*)
Cut tiles to a given layout and merge overlapping tiles. This will produce unique keys.

Parameters

- **layout** (*LayoutDefinition* or *Metadata* or TiledRasterLayer or *GlobalLayout* or *LocalLayout*) – Target raster layout for the tiling operation.
- **target_crs** (*str or int, optional*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string. If None, no reproject will be perfomed.
- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use for the reprojection. If none is specified, then ResampleMethods.NEAREST_NEIGHBOR is used.
- **partition_strategy** (*HashPartitionStrategy* or SpatialPartitioinStrategy or *SpaceTimePartitionStrategy*, optional) – Sets the Partitioner for the resulting layer and how many partitions it has. Default is, None.

If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *`TiledRasterLayer`*

`to_geotiff_rdd`(*storage_method=<StorageMethod.STRIPED: 'Striped'>*, *rows_per_strip=None*, *tile_dimensions=(256, 256)*, *compression=<Compression.NO_COMPRESSION: 'NoCompression'>*, *color_space=<ColorSpace.BLACK_IS_ZERO: 1>*, *color_map=None*, *head_tags=None*, *band_tags=None*)

Converts the rasters within this layer to GeoTiffs which are then converted to bytes. This is returned as a `RDD[(K, bytes)]`. Where `K` is either `SpatialKey` or `SpaceTimeKey`.

Parameters

- **`storage_method`** (str or *`StorageMethod`*, optional) – How the segments within the GeoTiffs should be arranged. Default is `StorageMethod.STRIPED`.
- **`rows_per_strip`** (*`int, optional`*) – How many rows should be in each strip segment of the GeoTiffs if `storage_method` is `StorageMethod.STRIPED`. If `None`, then the strip size will default to a value that is 8K or less.
- **`tile_dimensions`** (*`(int, int), optional`*) – The length and width for each tile segment of the GeoTiff if `storage_method` is `StorageMethod.TILED`. If `None` then the default size is `(256, 256)`.
- **`compression`** (str or *`Compression`*, optional) – How the data should be compressed. Defaults to `Compression.NO_COMPRESSION`.
- **`color_space`** (str or *`ColorSpace`*, optional) – How the colors should be organized in the GeoTiffs. Defaults to `ColorSpace.BLACK_IS_ZERO`.
- **`color_map`** (*`ColorMap`*, optional) – A `ColorMap` instance used to color the GeoTiffs to a different gradient.
- **`head_tags`** (*`dict, optional`*) – A `dict` where each key and value is a `str`.
- **`band_tags`** (*`list, optional`*) – A `list` of `dict`s where each key and value is a `str`.
- **`Note`** – For more information on the contents of the tags, see www.gdal.org/gdal_datamodel.html

Returns RDD[(K, bytes)]

`to_numpy_rdd`()

Converts a `TiledRasterLayer` to a numpy RDD.

Note: Depending on the size of the data stored within the RDD, this can be an exspensive operation and should be used with caution.

Returns RDD

to_png_rdd(*color_map*)
Converts the rasters within this layer to PNGs which are then converted to bytes. This is returned as a RDD[(K, bytes)].

Parameters **color_map** (*ColorMap*) – A `ColorMap` instance used to color the PNGs.

Returns RDD[(K, bytes)]

to_spatial_layer(*target_time=None*)
Converts a `TiledRasterLayer` with a `layout_type` of `LayoutType.SPACETIME` to a `TiledRasterLayer` with a `layout_type` of `LayoutType.SPATIAL`.

Parameters **target_time** (`datetime.datetime`, optional) – The instance of interest. If set, the resulting `TiledRasterLayer` will only contain keys that contained the given instance. If `None`, then all values within the layer will be kept.

Returns *TiledRasterLayer*

Raises `ValueError` – If the layer already has a `layout_type` of `LayoutType.SPATIAL`.

tobler()
Generates a Tobler walking speed layer from an elevation layer.

Note: This method has a known issue where the Tobler calculation is direction agnostic. Thus, all slopes are assumed to be uphill. This can result it incorrect results. A fix is currently being worked on.

Returns *TiledRasterLayer*

unpersist()
Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

with_no_data(*no_data_value*)
Changes the `NoData` value of the layer with the new given value.

It is possible to specify a `NoData` value for layers with raw values. The resulting layer will be of the same `CellType` but with a user defined `NoData` value. For example, if a layer has a `CellType` of `float32raw` and a `no_data_value` of `-10` is given, then the produced layer will have a `CellType` of `float32ud-10.0`.

If the target layer has a `bool` `CellType`, then the `no_data_value` will be ignored and the result layer will be the same as the origin. In order to assign a `NoData` value to a `bool` layer, the *convert_data_type()* method must be used.

Parameters **no_data_value** (*int or float*) – The new `NoData` value of the layer.

Returns *TiledRasterLayer*

wrapped_rdds()
Returns the list of RDD-containing objects wrapped by this object. The default implementation assumes that subclass contains a single RDD container, srdd, which implements the persist() and unpersist() methods.

class geopyspark.**Pyramid**(*levels*)
Contains a list of `TiledRasterLayers` that make up a tile pyramid. Each layer represents a level within the pyramid. This class is used when creating a tile server.

Map algebra can performed on instances of this class.

Parameters **levels** (*list or dict*) – A list of TiledRasterLayers or a dict of TiledRasterLayers where the value is the layer itself and the key is its given zoom level.

pysc
pyspark.SparkContext – The SparkContext being used this session.

layer_type (class
~geopyspark.geotrellis.constants.LayerType): What the layer type of the geotiffs are.

levels
dict – A dict of TiledRasterLayers where the value is the layer itself and the key is its given zoom level.

max_zoom
int – The highest zoom level of the pyramid.

is_cached
bool – Signals whether or not the internal RDDs are cached. Default is False.

histogram
Histogram – The Histogram that represents the layer with the max zoomw. Will not be calculated unless the *get_histogram()* method is used. Otherwise, its value is None.

Raises TypeError – If levels is neither a list or dict.

cache()
Persist this RDD with the default storage level (C{MEMORY_ONLY}).

count()
Returns how many elements are within the wrapped RDD.

Returns The number of elements in the RDD.

Return type Int

getNumPartitions()
Returns the number of partitions set for the wrapped RDD.

Returns The number of partitions.

Return type Int

get_histogram()
Calculates the Histogram for the layer with the max zoom.

Returns *Histogram*

get_partition_strategy()
Returns the partitioning strategy if the layer has one.

Returns HashPartitioner or SpatialPartitioner or *SpaceTimePartitionStrategy* or None

histogram

isEmpty()
Returns a bool that is True if the layer is empty and False if it is not.

Returns Are there elements within the layer

Return type bool

is_cached

layer_type

levels

max_zoom

persist (*storageLevel=StorageLevel(False, True, False, False, 1)*)
Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (C{MEMORY_ONLY}).

pysc

unpersist()
Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

wrapped_rdds()
Returns a list of the wrapped, Scala RDDs within each layer of the pyramid.

Returns [org.apache.spark.rdd.RDD]

class geopyspark.**Square**(*extent*)

class geopyspark.**Circle**(*radius*)
A circle neighborhood.

Parameters radius (*int or float*) – The radius of the circle that determines which cells fall within the bounding box.

radius
int or float – The radius of the circle that determines which cells fall within the bounding box.

param_1
float – Same as `radius`.

param_2
float – Unused param for `Circle`. Is 0.0.

param_3
float – Unused param for `Circle`. Is 0.0.

name
str – The name of the neighborhood which is, "circle".

Note: Cells that lie exactly on the radius of the circle are apart of the neighborhood.

class geopyspark.**Wedge**(*radius, start_angle, end_angle*)
A wedge neighborhood.

Parameters

- **radius** (*int or float*) – The radius of the wedge.
- **start_angle** (*int or float*) – The starting angle of the wedge in degrees.
- **end_angle** (*int or float*) – The ending angle of the wedge in degrees.

radius
int or float – The radius of the wedge.

start_angle
int or float – The starting angle of the wedge in degrees.

end_angle
int or float – The ending angle of the wedge in degrees.

param_1
float – Same as `radius`.

param_2
float – Same as `start_angle`.

param_3
float – Same as `end_angle`.

name
str – The name of the neighborhood which is, "wedge".

class geopyspark.**Nesw**(*extent*)
A neighborhood that includes a column and row intersection for the focus.

Parameters extent (*int or float*) – The extent of this neighborhood. This represents the how many cells past the focus the bounding box goes.

extent
int or float – The extent of this neighborhood. This represents the how many cells past the focus the bounding box goes.

param_1
float – Same as `extent`.

param_2
float – Unused param for `Nesw`. Is 0.0.

param_3
float – Unused param for `Nesw`. Is 0.0.

name
str – The name of the neighborhood which is, "nesw".

class geopyspark.**Annulus**(*inner_radius*, *outer_radius*)
An Annulus neighborhood.

Parameters

- **inner_radius** (*int or float*) – The radius of the inner circle.
- **outer_radius** (*int or float*) – The radius of the outer circle.

inner_radius
int or float – The radius of the inner circle.

outer_radius
int or float – The radius of the outer circle.

param_1
float – Same as `inner_radius`.

param_2
float – Same as `outer_radius`.

param_3
float – Unused param for `Annulus`. Is 0.0.

name
str – The name of the neighborhood which is, "annulus".

geopyspark.**rasterize**(*geoms*, *crs*, *zoom*, *fill_value*, *cell_type=<CellType.FLOAT64: 'float64'>*, *options=None*, *partition_strategy=None*)
Rasterizes a Shapely geometries.

Parameters

- **geoms** (*[shapely.geometry] or (shapely.geometry) or pyspark.RDD[shapely.geometry]*) – Either a list, tuple, or a Python RDD of shapely geometries to rasterize.
- **crs** (*str or int*) – The CRS of the input geometry.
- **zoom** (*int*) – The zoom level of the output raster.
- **fill_value** (*int or float*) – Value to burn into pixels intersectiong geometry
- **cell_type** (str or *CellType*) – Which data type the cells should be when created. Defaults to `CellType.FLOAT64`.
- **options** (*RasterizerOptions*, optional) – Pixel intersection options.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy`, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will have the default `Partitioner` and a number of paritions that was determined by the method.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

`geopyspark.`**`rasterize_features`**(*features*, *crs*, *zoom*, *cell_type=<CellType.FLOAT64: 'float64'>*, *options=None*, *zindex_cell_type=<CellType.INT8: 'int8'>*, *partition_strategy=None*)

Rasterizes a collection of *Feature*s.

Parameters

- **features** (*pyspark.RDD[Feature]*) – A Python `RDD` that contains *Feature*s.

  Note: The `properties` of each `Feature` must be an instance of *CellValue*.

- **crs** (*str or int*) – The CRS of the input geometry.
- **zoom** (*int*) – The zoom level of the output raster.

  Note: Not all rasterized `Features` may be present in the resulting layer if the `zoom` is not high enough.

- **cell_type** (str or *CellType*) – Which data type the cells should be when created. Defaults to `CellType.FLOAT64`.
- **options** (*RasterizerOptions*, optional) – Pixel intersection options.
- **zindex_cell_type** (str or *CellType*) – Which data type the `Z-Index` cells are. Defaults to `CellType.INT8`.

- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy`, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will have the default `Partitioner` and a number of paritions that was determined by the method.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

class geopyspark.**TileRender**(*render_function*)

A Python implementation of the Scala geopyspark.geotrellis.tms.TileRender interface. Permits a callback from Scala to Python to allow for custom rendering functions.

Parameters **render_function** (*Tile => PIL.Image.Image*) – A function to convert geopyspark.geotrellis.Tile to a PIL Image.

render_function

Tile => PIL.Image.Image – A function to convert geopyspark.geotrellis.Tile to a PIL Image.

class Java

implements = ['geopyspark.geotrellis.tms.TileRender']

renderEncoded(*scala_array*)

A function to convert an array to an image.

Parameters **scala_array** – A linear array of bytes representing the protobuf-encoded contents of a tile

Returns bytes representing an image

requiresEncoding()

class geopyspark.**TMS**(*server*)

Provides a TMS server for raster data.

In order to display raster data on a variety of different map interfaces (e.g., leaflet maps, geojson.io, GeoNotebook, and others), we provide the TMS class.

Parameters **server** (*JavaObject*) – The Java TMSServer instance

pysc

pyspark.SparkContext – The `SparkContext` being used this session.

server

JavaObject – The Java TMSServer instance

host

str – The IP address of the host, if bound, else None

port

int – The port number of the TMS server, if bound, else None

url_pattern

string – The URI pattern for the current TMS service, with {z}, {x}, {y} tokens. Can be copied directly to services such as *geojson.io*.

bind(*host=None*, *requested_port=None*)

Starts up a TMS server.

Parameters

- **host** (*str, optional*) – The target host. Typically “localhost”, “127.0.0.1”, or “0.0.0.0”. The latter will make the TMS service accessible from the world. If omitted, defaults to localhost.
- **requested_port** (*optional, int*) – A port number to bind the service to. If omitted, use a random available port.

classmethod build(*source*, *display*, *allow_overzooming=True*)

Builds a TMS server from one or more layers.

This function takes a SparkContext, a source or list of sources, and a display method and creates a TMS server to display the desired content. The display method is supplied as a ColorMap (only available when there is a single source), or a callable object which takes either a single tile input (when there is a single source) or a list of tiles (for multiple sources) and returns the bytes representing an image file for that tile.

Parameters

- **source** (tuple or orlist or *Pyramid*) – The tile sources to render. Tuple inputs are (str, str) pairs where the first component is the URI of a catalog and the second is the layer name. A list input may be any combination of tuples and `Pyramid`s.
- **display** (`ColorMap`, *callable*) – Method for mapping tiles to images. ColorMap may only be applied to single input source. Callable will take a single numpy array for a single source, or a list of numpy arrays for multiple sources. In the case of multiple inputs, resampling may be required if the tile sources have different tile sizes. Returns bytes representing the resulting image.
- **allow_overzooming** (*bool*) – If set, viewing at zoom levels above the highest available zoom level will produce tiles that are resampled from the highest zoom level present in the data set.

host

Returns the IP string of the server’s host if bound, else None.

Returns (str)

port

Returns the port number for the current TMS server if bound, else None.

Returns (int)

set_handshake(*handshake*)

unbind()

Shuts down the TMS service, freeing the assigned port.

url_pattern

Returns the URI for the tiles served by the present server. Contains {z}, {x}, and {y} tokens to be substituted for the desired zoom and x/y tile position.

Returns (str)

`geopyspark.`**union**(*layers*)

Unions togther two or more `RasterLayer`s or `TiledRasterLayer`s.

All layers must have the same `layer_type`. If the layers are `TiledRasterLayer`s, then all of the layers must also have the same *TileLayout* and `CRS`.

Note: If the layers to be unioned share one or more keys, then the resulting layer will contain duplicates of that key. One copy for each instance of the key.

Parameters **layers** ([*RasterLayer*] or [*TiledRasterLayer*] or (*RasterLayer*) or (*TiledRasterLayer*)) – A colection of two or more `RasterLayers` or `TiledRasterLayers` layers to be unioned together.

Returns *RasterLayer* or *TiledRasterLayer*

`geopyspark.`**`combine_bands`**`(`*layers*`)`

Combines the bands of values that share the same key in two or more `TiledRasterLayers`.

This method will concat the bands of two or more values with the same key. For example, `layer a` has values that have 2 bands and `layer b` has values with 1 band. When `combine_bands` is used on both of these layers, then the resulting layer will have values with 3 bands, 2 from `layer a` and 1 from `layer b`.

Note: All layers must have the same `layer_type`. If the layers are `TiledRasterLayers`, then all of the layers must also have the same *TileLayout* and `CRS`.

Parameters **layers** ([*RasterLayer*] or [*TiledRasterLayer*] or (*RasterLayer*) or (*TiledRasterLayer*)) – A colection of two or more `RasterLayers` or `TiledRasterLayers`. **The order of the layers determines the order in which the bands are concatenated**. With the bands being ordered based on the position of their respective layer.

For example, the first layer in `layers` is `layer a` which contains 2 bands and the second layer is `layer b` whose values have 1 band. The resulting layer will have values with 3 bands: the first 2 are from `layer a` and the third from `layer b`. If the positions of `layer a` and `layer b` are reversed, then the resulting values' first band will be from `layer b` and the last 2 will be from `layer a`.

Returns *RasterLayer* or *TiledRasterLayer*

`class` `geopyspark.`**`Feature`**

Represents a geometry that is derived from an OSM Element with that Element's associated metadata.

Parameters

- **geometry** (*shapely.geometry*) – The geometry of the feature that is represented as a `shapely.geometry`. This geometry is derived from an OSM Element.
- **properties** (*Properties* or *CellValue*) – The metadata associated with the OSM Element. Can be represented as either an instance of `Properties` or a `CellValue`.

`geometry`

shapely.geometry – The geometry of the feature that is represented as a `shapely.geometry`. This geometry is derived from an OSM Element.

`properties`

Properties or *CellValue* – The metadata associated with the OSM Element. Can be represented as either an instance of `Properties` or a `CellValue`.

`count``(`*value*`)` $\rightarrow$ integer – return number of occurrences of value

`geometry`

Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

properties
Alias for field number 1

class geopyspark.**Properties**
Represents the metadata of an OSM Element.

This object is one of two types that can be used to represent the `properties` of a *Feature*.

Parameters

- **element_id** (*int*) – The `id` of the OSM Element.
- **user** (*str*) – The display name of the last user who modified/created the OSM Element.
- **uid** (*int*) – The numeric id of the last user who modified the OSM Element.
- **changeset** (*int*) – The OSM `changeset` number in which the OSM Element was created/modified.
- **version** (*int*) – The edit version of the OSM Element.
- **minor_version** (*int*) – Represents minor changes between versions of an OSM Element.
- **timestamp** (*datetime.datetime*) – The time of the last modification to the OSM Element.
- **visible** (*bool*) – Represents whether or not the OSM Element is deleted or not in the database.
- **tags** (*dict*) – A `dict` of `str`s that represents the given features of the OSM Element.

element_id
int – The `id` of the OSM Element.

user
str – The display name of the last user who modified/created the OSM Element.

uid
int – The numeric id of the last user who modified the OSM Element.

changeset
int – The OSM `changeset` number in which the OSM Element was created/modified.

version
int – The edit version of the OSM Element.

minor_version
int – Represents minor changes between versions of an OSM Element.

timestamp
datetime.datetime – The time of the last modification to the OSM Element.

visible
bool – Represents whether or not the OSM Element is deleted or not in the database.

tags
dict – A `dict` of `str`s that represents the given features of the OSM Element.

changeset
Alias for field number 3

count(*value*) → integer – return number of occurrences of value

element_id
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

minor_version
Alias for field number 5

tags
Alias for field number 8

timestamp
Alias for field number 6

uid
Alias for field number 2

user
Alias for field number 1

version
Alias for field number 4

visible
Alias for field number 7

class geopyspark.**CellValue**
Represents the `value` and `zindex` of a geometry.

This object is one of two types that can be used to represent the `properties` of a *`Feature`*.

Parameters

- **value** (*int or float*) – The value of all cells that intersects the associated geometry.
- **zindex** (*int*) – The `Z-Index` of each cell that intersects the associated geometry. `Z-Index` determines which value a cell should be if multiple geometries intersect it. A high `Z-Index` will always be in front of a `Z-Index` of a lower value.

value
int or float – The value of all cells that intersects the associated geometry.

zindex
int – The `Z-Index` of each cell that intersects the associated geometry. `Z-Index` determines which value a cell should be if multiple geometries intersect it. A high `Z-Index` will always be in front of a `Z-Index` of a lower value.

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

value
Alias for field number 0

zindex
Alias for field number 1

2.13 geopyspark.geotrellis package

class `geopyspark.geotrellis.`**`Tile`**

Represents a raster in GeoPySpark.

Note: All rasters in GeoPySpark are represented as having multiple bands, even if the original raster just contained one.

Parameters

- **`cells`** (*`nd.array`*) – The raster data itself. It is contained within a NumPy array.
- **`data_type`** (*`str`*) – The data type of the values within `data` if they were in Scala.
- **`no_data_value`** – The value that represents no data value in the raster. This can be represented by a variety of types depending on the value type of the raster.

`cells`

nd.array – The raster data itself. It is contained within a NumPy array.

`data_type`

str – The data type of the values within `data` if they were in Scala.

`no_data_value`

The value that represents no data value in the raster. This can be represented by a variety of types depending on the value type of the raster.

`cell_type`

Alias for field number 1

`cells`

Alias for field number 0

`count`(*value*) → integer – return number of occurrences of value

`static dtype_to_cell_type`(*dtype*)

Converts a `np.dtype` to the corresponding GeoPySpark `cell_type`.

Note: `bool`, `complex64`, `complex128`, and `complex256`, are currently not supported `np.dtypes`.

Parameters **`dtype`** (*`np.dtype`*) – The `dtype` of the numpy array.

Returns str. The GeoPySpark `cell_type` equivalent of the `dtype`.

Raises `TypeError` – If the given `dtype` is not a supported data type.

`classmethod from_numpy_array`(*numpy_array*, *no_data_value=None*)

Creates an instance of `Tile` from a numpy array.

Parameters

- **`numpy_array`** (*`np.array`*) – The numpy array to be used to represent the cell values of the `Tile`.

Note: GeoPySpark does not support arrays with the following data types: `bool`, `complex64`, `complex128`, and `complex256`.

- **no_data_value** (*optional*) – The value that represents no data value in the raster. This can be represented by a variety of types depending on the value type of the raster. If not given, then the value will be `None`.

Returns *Tile*

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

no_data_value
Alias for field number 2

class geopyspark.geotrellis.**Extent**
The "bounding box" or geographic region of an area on Earth a raster represents.

Parameters

- **xmin** (*float*) – The minimum x coordinate.
- **ymin** (*float*) – The minimum y coordinate.
- **xmax** (*float*) – The maximum x coordinate.
- **ymax** (*float*) – The maximum y coordinate.

xmin
float – The minimum x coordinate.

ymin
float – The minimum y coordinate.

xmax
float – The maximum x coordinate.

ymax
float – The maximum y coordinate.

count(*value*) → integer – return number of occurrences of value

classmethod from_polygon(*polygon*)
Creates a new instance of `Extent` from a Shapely Polygon.

The new `Extent` will contain the min and max coordinates of the Polygon; regardless of the Polygon's shape.

Parameters polygon (*shapely.geometry.Polygon*) – A Shapely Polygon.

Returns *Extent*

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

to_polygon
Converts this instance to a Shapely Polygon.

The resulting Polygon will be in the shape of a box.

Returns `shapely.geometry.Polygon`

xmax
Alias for field number 2

xmin
Alias for field number 0

ymax
Alias for field number 3

ymin
Alias for field number 1

class geopyspark.geotrellis.**ProjectedExtent**
Describes both the area on Earth a raster represents in addition to its CRS.

Parameters

- **extent** (*Extent*) – The area the raster represents.
- **epsg** (*int, optional*) – The EPSG code of the CRS.
- **proj4** (*str, optional*) – The Proj.4 string representation of the CRS.

extent
Extent – The area the raster represents.

epsg
int, optional – The EPSG code of the CRS.

proj4
str, optional – The Proj.4 string representation of the CRS.

Note: Either `epsg` or `proj4` must be defined.

count(*value*) → integer – return number of occurrences of value

epsg
Alias for field number 1

extent
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

proj4
Alias for field number 2

class geopyspark.geotrellis.**TemporalProjectedExtent**
Describes the area on Earth the raster represents, its CRS, and the time the data was collected.

Parameters

- **extent** (*Extent*) – The area the raster represents.
- **instant** (`datetime.datetime`) – The time stamp of the raster.
- **epsg** (*int, optional*) – The EPSG code of the CRS.
- **proj4** (*str, optional*) – The Proj.4 string representation of the CRS.

extent
Extent – The area the raster represents.

instant
`datetime.datetime` – The time stamp of the raster.

epsg
int, optional – The EPSG code of the CRS.

proj4
str, optional – The Proj.4 string representation of the CRS.

Note: Either `epsg` or `proj4` must be defined.

---

count (*value*) → integer – return number of occurrences of value

epsg
Alias for field number 2

extent
Alias for field number 0

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

instant
Alias for field number 1

proj4
Alias for field number 3

class geopyspark.geotrellis.**GlobalLayout**
TileLayout type that spans global CRS extent.

When passed in place of LayoutDefinition it signifies that a LayoutDefinition instance should be constructed such that it fits the global CRS extent. The cell resolution of resulting layout will be one of resolutions implied by power of 2 pyramid for that CRS. Tiling to this layout will likely result in either up-sampling or down-sampling the source raster.

Parameters

- **tile_size** (*int*) – The number of columns and row pixels in each tile.
- **zoom** (*int, optional*) – Override the zoom level in power of 2 pyramid.
- **threshold** (*float, optional*) – The percentage difference between a cell size and a zoom level and the resolution difference between that zoom level and the next that is tolerated to snap to the lower-resolution zoom level. For example, if this paramter is 0.1, that means we're willing to downsample rasters with a higher resolution in order to fit them to some zoom level Z, if the difference is resolution is less than or equal to 10% the difference between the resolutions of zoom level Z and zoom level Z+1.

tile_size
int – The number of columns and row pixels in each tile.

zoom
int – The desired zoom level of the layout.

threshold
float, optional – The percentage difference between a cell size and a zoom level and the resolution difference between that zoom level and the next that is tolerated to snap to the lower-resolution zoom level.

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

threshold
Alias for field number 2

tile_size
Alias for field number 0

zoom
Alias for field number 1

class geopyspark.geotrellis.**LocalLayout**
TileLayout type that snaps the layer extent.

When passed in place of LayoutDefinition it signifies that a LayoutDefinition instances should be constructed over the envelope of the layer pixels with given tile size. Resulting TileLayout will match the cell resolution of the source rasters.

Parameters

- **tile_size** (*int, optional*) – The number of columns and row pixels in each tile. If this is `None`, then the sizes of each tile will be set using `tile_cols` and `tile_rows`.
- **tile_cols** (*int, optional*) – The number of column pixels in each tile. This supersedes `tile_size`. Meaning if this and `tile_size` are set, then this will be used for the number of colunn pixles. If `None`, then the number of column pixels will default to 256.
- **tile_rows** (*int, optional*) – The number of rows pixels in each tile. This supersedes `tile_size`. Meaning if this and `tile_size` are set, then this will be used for the number of row pixles. If `None`, then the number of row pixels will default to 256.

tile_cols
int – The number of column pixels in each tile

tile_rows
int – The number of rows pixels in each tile. This supersedes

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

tile_cols
Alias for field number 0

tile_rows
Alias for field number 1

class geopyspark.geotrellis.**LocalLayout**
TileLayout type that snaps the layer extent.

When passed in place of LayoutDefinition it signifies that a LayoutDefinition instances should be constructed over the envelope of the layer pixels with given tile size. Resulting TileLayout will match the cell resolution of the source rasters.

Parameters

- **tile_size** (*int, optional*) – The number of columns and row pixels in each tile. If this is `None`, then the sizes of each tile will be set using `tile_cols` and `tile_rows`.
- **tile_cols** (*int, optional*) – The number of column pixels in each tile. This supersedes `tile_size`. Meaning if this and `tile_size` are set, then this will be used for the number of colunn pixles. If `None`, then the number of column pixels will default to 256.

- **tile_rows** (*int, optional*) – The number of rows pixels in each tile. This supersedes `tile_size`. Meaning if this and `tile_size` are set, then this will be used for the number of row pixles. If `None`, then the number of row pixels will default to 256.

tile_cols
int – The number of column pixels in each tile

tile_rows
int – The number of rows pixels in each tile. This supersedes

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

tile_cols
Alias for field number 0

tile_rows
Alias for field number 1

class geopyspark.geotrellis.**TileLayout**
Describes the grid in which the rasters within a Layer should be laid out.

Parameters

- **layoutCols** (*int*) – The number of columns of rasters that runs east to west.
- **layoutRows** (*int*) – The number of rows of rasters that runs north to south.
- **tileCols** (*int*) – The number of columns of pixels in each raster that runs east to west.
- **tileRows** (*int*) – The number of rows of pixels in each raster that runs north to south.

layoutCols
int – The number of columns of rasters that runs east to west.

layoutRows
int – The number of rows of rasters that runs north to south.

tileCols
int – The number of columns of pixels in each raster that runs east to west.

tileRows
int – The number of rows of pixels in each raster that runs north to south.

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

layoutCols
Alias for field number 0

layoutRows
Alias for field number 1

tileCols
Alias for field number 2

tileRows
Alias for field number 3

class geopyspark.geotrellis.**LayoutDefinition**
Describes the layout of the rasters within a Layer and how they are projected.

Parameters

- **extent** (*Extent*) – The Extent of the layout.
- **tileLayout** (*TileLayout*) – The TileLayout of how the rasters within the Layer.

extent
Extent – The Extent of the layout.

tileLayout
TileLayout – The TileLayout of how the rasters within the Layer.

count(*value*) → integer – return number of occurrences of value

extent
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

tileLayout
Alias for field number 1

class geopyspark.geotrellis.**SpatialKey**
Represents the position of a raster within a grid. This grid is a 2D plane where raster positions are represented by a pair of coordinates.

Parameters

- **col** (*int*) – The column of the grid, the numbers run east to west.
- **row** (*int*) – The row of the grid, the numbers run north to south.

col
int – The column of the grid, the numbers run east to west.

row
int – The row of the grid, the numbers run north to south.

col
Alias for field number 0

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

row
Alias for field number 1

class geopyspark.geotrellis.**SpaceTimeKey**
Represents the position of a raster within a grid. This grid is a 3D plane where raster positions are represented by a pair of coordinates as well as a z value that represents time.

Parameters

- **col** (*int*) – The column of the grid, the numbers run east to west.
- **row** (*int*) – The row of the grid, the numbers run north to south.
- **instant** (datetime.datetime) – The time stamp of the raster.

col
int – The column of the grid, the numbers run east to west.

row
int – The row of the grid, the numbers run north to south.

instant
`datetime.datetime` – The time stamp of the raster.

col
Alias for field number 0

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

instant
Alias for field number 2

row
Alias for field number 1

class geopyspark.geotrellis.**RasterizerOptions**
Represents options available to geometry rasterizer

Parameters

- **includePartial** (*bool, optional*) – Include partial pixel intersection (default: True)
- **sampleType** (*str, optional*) – 'PixelIsArea' or 'PixelIsPoint' (default: 'PixelIsPoint')

includePartial
bool – Include partial pixel intersection.

sampleType
str – How the sampling should be performed during rasterization.

count (*value*) → integer – return number of occurrences of value

includePartial
Alias for field number 0

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

sampleType
Alias for field number 1

class geopyspark.geotrellis.**Bounds**
Represents the grid that covers the area of the rasters in a Layer on a grid.

Parameters

- **minKey** (*SpatialKey* or *SpaceTimeKey*) – The smallest `SpatialKey` or `SpaceTimeKey`.
- **minKey** – The largest `SpatialKey` or `SpaceTimeKey`.

minKey
SpatialKey or *SpaceTimeKey* – The smallest `SpatialKey` or `SpaceTimeKey`.

minKey
SpatialKey or *SpaceTimeKey* – The largest `SpatialKey` or `SpaceTimeKey`.

count (*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

maxKey
Alias for field number 1

minKey
Alias for field number 0

class geopyspark.geotrellis.**Metadata**(*bounds*, *crs*, *cell_type*, *extent*, *layout_definition*)
Information of the values within a `RasterLayer` or `TiledRasterLayer`. This data pertains to the layout and other attributes of the data within the classes.

Parameters

- **bounds** (*Bounds*) – The `Bounds` of the values in the class.
- **crs** (*str or int*) – The `CRS` of the data. Can either be the EPSG code, well-known name, or a PROJ.4 projection string.
- **cell_type** (str or *CellType*) – The data type of the cells of the rasters.
- **extent** (*Extent*) – The `Extent` that covers the all of the rasters.
- **layout_definition** (*LayoutDefinition*) – The `LayoutDefinition` of all rasters.

bounds
Bounds – The `Bounds` of the values in the class.

crs
str or int – The CRS of the data. Can either be the EPSG code, well-known name, or a PROJ.4 projection string.

cell_type
str – The data type of the cells of the rasters.

no_data_value
int or float or None – The noData value of the rasters within the layer. This can either be `None`, an `int`, or a `float` depending on the `cell_type`.

extent
Extent – The `Extent` that covers the all of the rasters.

tile_layout
TileLayout – The `TileLayout` that describes how the rasters are orginized.

layout_definition
LayoutDefinition – The `LayoutDefinition` of all rasters.

classmethod from_dict(*metadata_dict*)
Creates `Metadata` from a dictionary.

Parameters metadata_dict (*dict*) – The `Metadata` of a `RasterLayer` or `TiledRasterLayer` instance that is in `dict` form.

Returns *Metadata*

to_dict()
Converts this instance to a `dict`.

Returns `dict`

2.13.1 geopyspark.geotrellis.catalog module

Methods for reading, querying, and saving tile layers to and from GeoTrellis Catalogs.

`geopyspark.geotrellis.catalog.`**`read_layer_metadata`**(*uri*, *layer_name*, *layer_zoom*)

Reads the metadata from a saved layer without reading in the whole layer.

Parameters

- **`uri`** (*`str`*) – The Uniform Resource Identifier used to point towards the desired GeoTrellis catalog to be read from. The shape of this string varies depending on backend.
- **`layer_name`** (*`str`*) – The name of the GeoTrellis catalog to be read from.
- **`layer_zoom`** (*`int`*) – The zoom level of the layer that is to be read.

Returns *`Metadata`*

`geopyspark.geotrellis.catalog.`**`read_value`**(*uri*, *layer_name*, *layer_zoom*, *col*, *row*, *zdt=None*)

Reads a single `Tile` from a GeoTrellis catalog. Unlike other functions in this module, this will not return a `TiledRasterLayer`, but rather a GeoPySpark formatted raster.

Note: When requesting a tile that does not exist, `None` will be returned.

Parameters

- **`uri`** (*`str`*) – The Uniform Resource Identifier used to point towards the desired GeoTrellis catalog to be read from. The shape of this string varies depending on backend.
- **`layer_name`** (*`str`*) – The name of the GeoTrellis catalog to be read from.
- **`layer_zoom`** (*`int`*) – The zoom level of the layer that is to be read.
- **`col`** (*`int`*) – The col number of the tile within the layout. Cols run east to west.
- **`row`** (*`int`*) – The row number of the tile within the layout. Row run north to south.
- **`zdt`** (`datetime.datetime`) – The time stamp of the tile if the data is spatial-temporal. This is represented as a `datetime.datetime.` instance. The default value is, `None`. If `None`, then only the spatial area will be queried.

Returns *`Tile`*

`geopyspark.geotrellis.catalog.`**`query`**(*uri*, *layer_name*, *layer_zoom=None*, *query_geom=None*, *time_intervals=None*, *query_proj=None*, *num_partitions=None*)

Queries a single, zoom layer from a GeoTrellis catalog given spatial and/or time parameters.

Note: The whole layer could still be read in if `intersects` and/or `time_intervals` have not been set, or if the querried region contains the entire layer.

Parameters

- **`layer_type`** (str or *`LayerType`*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **`uri`** (*`str`*) – The Uniform Resource Identifier used to point towards the desired GeoTrellis catalog to be read from. The shape of this string varies depending on backend.

- **layer_name** (*str*) – The name of the GeoTrellis catalog to be querried.
- **layer_zoom** (*int, optional*) – The zoom level of the layer that is to be querried. If `None`, then the `layer_zoom` will be set to 0.
- **query_geom** (bytes or shapely.geometry or *Extent*, Optional) – The desired spatial area to be returned. Can either be a string, a shapely geometry, or instance of `Extent`, or a WKB verson of the geometry.

  Note: Not all shapely geometires are supported. The following is are the types that are supported: * Point * Polygon * MultiPolygon

  Note: Only layers that were made from spatial, singleband GeoTiffs can query a `Point`. All other types are restricted to `Polygon` and `MulitPolygon`.

  Note: If the queried region does not intersect the layer, then an empty layer will be returned.

  If not specified, then the entire layer will be read.
- **time_intervals** (`[datetime.datetime]`, optional) – A list of the time intervals to query. This parameter is only used when querying spatial-temporal data. The default value is, `None`. If `None`, then only the spatial area will be querried.
- **query_proj** (*int or str, optional*) – The crs of the querried geometry if it is different than the layer it is being filtered against. If they are different and this is not set, then the returned `TiledRasterLayer` could contain incorrect values. If `None`, then the geometry and layer are assumed to be in the same projection.
- **num_partitions** (*int, optional*) – Sets RDD partition count when reading from catalog.

Returns *TiledRasterLayer*

`geopyspark.geotrellis.catalog.`**`write`**(*uri*, *layer_name*, *tiled_raster_layer*, *index_strategy=<IndexingMethod.ZORDER: 'zorder'>*, *time_unit=None*, *time_resolution=None*, *store=None*)

Writes a tile layer to a specified destination.

Parameters

- **uri** (*str*) – The Uniform Resource Identifier used to point towards the desired location for the tile layer to written to. The shape of this string varies depending on backend.
- **layer_name** (*str*) – The name of the new, tile layer.
- **layer_zoom** (*int*) – The zoom level the layer should be saved at.
- **tiled_raster_layer** (*TiledRasterLayer*) – The `TiledRasterLayer` to be saved.
- **index_strategy** (str or *IndexingMethod*) – The method used to orginize the saved data. Depending on the type of data within the layer, only certain methods are available. Can either be a string or a `IndexingMethod` attribute. The default method used is, `IndexingMethod.ZORDER`.

- **time_unit** (str or *TimeUnit*, optional) – Which time unit should be used when saving spatial-temporal data. This controls the resolution of each index. Meaning, what time intervals are used to seperate each record. While this is set to `None` as default, it must be set if saving spatial-temporal data. Depending on the indexing method chosen, different time units are used.
- **time_resolution** (*str or int, optional*) – Determines how data for each `time_unit` should be grouped together. By default, no grouping will occur.

  As an example, having a `time_unit` of `WEEKS` and a `time_resolution` of 5 will cause the data to be grouped and stored together in units of 5 weeks. If however `time_resolution` is not specified, then the data will be grouped and stored in units of single weeks.

  This value can either be an `int` or a string representation of an `int`.
- **store** (str or *AttributeStore*, optional) – `AttributeStore` instance or URI for layer metadata lookup.

class geopyspark.geotrellis.catalog.**AttributeStore**(*uri*)

AttributeStore provides a way to read and write GeoTrellis layer attributes.

Internally all attribute values are stored as JSON, here they are exposed as dictionaries. Classes often stored have a `.from_dict` and `.to_dict` methods to bridge the gap:

```
import geopyspark as gps
store = gps.AttributeStore("s3://azavea-datahub/catalog")
hist = store.layer("us-nlcd2011-30m-epsg3857", zoom=7).read("histogram")
hist = gps.Histogram.from_dict(hist)
```

class Attributes(*store, layer_name, layer_zoom*)

Accessor class for all attributes for a given layer

delete(*name*)

Delete attribute by name

Parameters name (*str*) – Attribute name

read(*name*)

Read layer attribute by name as a dict

Parameters name (*str*) –

Returns Attribute value

Return type `dict`

write(*name, value*)

Write layer attribute value as a dict

Parameters

- **name** (*str*) – Attribute name
- **value** (*dict*) – Attribute value

classmethod build(*store*)

Builds AttributeStore from URI or passes an instance through.

Parameters uri (*str or AttributeStore*) – URI for AttributeStore object or instance.

Returns *AttributeStore*

classmethod cached(*uri*)

Returns cached version of AttributeStore for URI or creates one

contains(*name, zoom=None*)

Checks if this store contains a layer metadata.

Parameters

- **name** (*str*) – Layer name
- **zoom** (*int, optional*) – Layer zoom

Returns `bool`

delete(*name, zoom=None*)
Delete layer and all its attributes

Parameters

- **name** (*str*) – Layer name
- **zoom** (*int, optional*) – Layer zoom

layer(*name, zoom=None*)
Layer Attributes object for given layer :param name: Layer name :type name: str :param zoom: Layer zoom :type zoom: int, optional

Returns `Attributes`

layers()
List all layers Attributes objects

Returns `[:class:`~geopyspark.geotrellis.catalog.AttributeStore.Attributes`]`

2.13.2 geopyspark.geotrellis.color module

This module contains functions needed to create color maps used in coloring tiles, PNGs, and GeoTiffs.

geopyspark.geotrellis.color.**get_colors_from_colors**(*colors*)
Returns a list of integer colors from a list of Color objects from the colortools package.

Parameters **colors** (*[colortools.Color]*) – A list of color stops using colortools.Color

Returns [int]

geopyspark.geotrellis.color.**get_colors_from_matplotlib**(*ramp_name, num_colors=256*)
Returns a list of color breaks from the color ramps defined by Matplotlib.

Parameters

- **ramp_name** (*str*) – The name of a matplotlib color ramp. See the matplotlib documentation for a list of names and details on each color ramp.
- **num_colors** (*int, optional*) – The number of color breaks to derive from the named map.

Returns [int]

class geopyspark.geotrellis.color.**ColorMap**(*cmap*)
A class that wraps a GeoTrellis ColorMap class.

Parameters **cmap** (*py4j.java_gateway.JavaObject*) – The `JavaObject` that represents the GeoTrellis ColorMap.

cmap
py4j.java_gateway.JavaObject – The `JavaObject` that represents the GeoTrellis ColorMap.

classmethod build(*breaks*, *colors=None*, *no_data_color=0*, *fallback=0*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)

Given breaks and colors, build a `ColorMap` object.

Parameters

- **breaks** (dict or list or `np.ndarray` or `Histogram`) – If a `dict` then a mapping from tile values to colors, the latter represented as integers e.g., 0xff000080 is red at half opacity. If a `list` then tile values that specify breaks in the color mapping. If a `Histogram` then a histogram from which breaks can be derived.
- **colors** (*str or list, optional*) – If a `str` then the name of a matplotlib color ramp. If a `list` then either a list of colortools `Color` objects or a list of integers containing packed RGBA values. If `None`, then the `ColorMap` will be created from the `breaks` given.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

classmethod from_break_map(*break_map*, *no_data_color=0*, *fallback=0*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)

Converts a dictionary mapping from tile values to colors to a ColorMap.

Parameters

- **break_map** (*dict*) – A mapping from tile values to colors, the latter represented as integers e.g., 0xff000080 is red at half opacity.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

classmethod from_colors(*breaks*, *color_list*, *no_data_color=0*, *fallback=0*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)

Converts lists of values and colors to a `ColorMap`.

Parameters

- **breaks** (*list*) – The tile values that specify breaks in the color mapping.

- **color_list** (*[int]*) – The colors corresponding to the values in the breaks list, represented as integers—e.g., 0xff000080 is red at half opacity.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

classmethod from_histogram(*histogram, color_list, no_data_color=0, fallback=0, classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*)

Converts a wrapped GeoTrellis histogram into a `ColorMap`.

Parameters

- **histogram** (`Histogram`) – A `Histogram` instance; specifies breaks
- **color_list** (*[int]*) – The colors corresponding to the values in the breaks list, represented as integers e.g., 0xff000080 is red at half opacity.
- **no_data_color** (*int, optional*) – A color to replace NODATA values with
- **fallback** (*int, optional*) – A color to replace cells that have no value in the mapping
- **classification_strategy** (str or *ClassificationStrategy*, optional) – A string giving the strategy for converting tile values to colors. e.g., if `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` is specified, and the break map is {3: 0xff0000ff, 4: 0x00ff00ff}, then values up to 3 map to red, values from above 3 and up to and including 4 become green, and values over 4 become the fallback color.

Returns *ColorMap*

static nlcd_colormap()

Returns a color map for NLCD tiles.

Returns *ColorMap*

2.13.3 geopyspark.geotrellis.combine_bands module

`geopyspark.geotrellis.combine_bands.`**combine_bands**(*layers*)

Combines the bands of values that share the same key in two or more `TiledRasterLayers`.

This method will concat the bands of two or more values with the same key. For example, `layer a` has values that have 2 bands and `layer b` has values with 1 band. When `combine_bands` is used on both of these layers, then the resulting layer will have values with 3 bands, 2 from `layer a` and 1 from `layer b`.

Note: All layers must have the same `layer_type`. If the layers are `TiledRasterLayers`, then all of the layers must also have the same *TileLayout* and `CRS`.

Parameters layers ([*RasterLayer*] or [*TiledRasterLayer*] or (*RasterLayer*) or (*TiledRasterLayer*)) – A colection of two or more `RasterLayers` or `TiledRasterLayers`. **The order of the layers determines the order in which the bands are concatenated**. With the bands being ordered based on the position of their respective layer.

For example, the first layer in `layers` is `layer a` which contains 2 bands and the second layer is `layer b` whose values have 1 band. The resulting layer will have values with 3 bands: the first 2 are from `layer a` and the third from `layer b`. If the positions of `layer a` and `layer b` are reversed, then the resulting values' first band will be from `layer b` and the last 2 will be from `layer a`.

Returns *RasterLayer* or *TiledRasterLayer*

2.13.4 geopyspark.geotrellis.constants module

Constants that are used by `geopyspark.geotrellis` classes, methods, and functions.

`geopyspark.geotrellis.constants.`**`NO_DATA_INT = -2147483648`**
The default size of each tile in the resulting layer.

class `geopyspark.geotrellis.constants.`**`LayerType`**
The type of the key within the tuple of the wrapped RDD.

`SPACETIME = 'spacetime'`

`SPATIAL = 'spatial'`
Indicates that the RDD contains `(K, V)` pairs, where the `K` has a spatial and time attribute. Both *TemporalProjectedExtent* and *SpaceTimeKey* are examples of this type of `K`.

class `geopyspark.geotrellis.constants.`**`IndexingMethod`**
How the wrapped should be indexed when saved.

`HILBERT = 'hilbert'`
A key indexing method. Works only for RDDs that contain *SpatialKey*. This method provides the fastest lookup of all the key indexing method, however, it does not give good locality guarantees. It is recommended then that this method should only be used when locality is not important for your analysis.

`ROWMAJOR = 'rowmajor'`

`ZORDER = 'zorder'`
A key indexing method. Works for RDDs that contain both *SpatialKey* and *SpaceTimeKey*. Note, indexes are determined by the `x`, `y`, and if `SPACETIME`, the temporal resolutions of a point. This is expressed in bits, and has a max value of 62. Thus if the sum of those resolutions are greater than 62, then the indexing will fail.

class `geopyspark.geotrellis.constants.`**`ResampleMethod`**
Resampling Methods.

`AVERAGE = 'Average'`

`BILINEAR = 'Bilinear'`

`CUBIC_CONVOLUTION = 'CubicConvolution'`

`CUBIC_SPLINE = 'CubicSpline'`

`LANCZOS = 'Lanczos'`

`MAX = 'Max'`

`MEDIAN = 'Median'`

`MIN = 'Min'`

MODE = 'Mode'

NEAREST_NEIGHBOR = 'NearestNeighbor'

class geopyspark.geotrellis.constants.**TimeUnit**

ZORDER time units.

DAYS = 'days'

HOURS = 'hours'

MILLIS = 'millis'

MINUTES = 'minutes'

MONTHS = 'months'

SECONDS = 'seconds'

WEEKS = 'weeks'

YEARS = 'years'

class geopyspark.geotrellis.constants.**Operation**

Focal opertions.

ASPECT = 'Aspect'

MAX = 'Max'

MEAN = 'Mean'

MEDIAN = 'Median'

MIN = 'Min'

MODE = 'Mode'

STANDARD_DEVIATION = 'StandardDeviation'

SUM = 'Sum'

VARIANCE = 'Variance'

class geopyspark.geotrellis.constants.**Neighborhood**

Neighborhood types.

ANNULUS = 'Annulus'

CIRCLE = 'Circle'

NESW = 'Nesw'

SQUARE = 'Square'

WEDGE = 'Wedge'

class geopyspark.geotrellis.constants.**ClassificationStrategy**

Classification strategies for color mapping.

EXACT = 'Exact'

GREATER_THAN = 'GreaterThan'

GREATER_THAN_OR_EQUAL_TO = 'GreaterThanOrEqualTo'

LESS_THAN = 'LessThan'

LESS_THAN_OR_EQUAL_TO = 'LessThanOrEqualTo'

class geopyspark.geotrellis.constants.**CellType**

Cell types.

BOOL = 'bool'

BOOLRAW = 'boolraw'

FLOAT32 = 'float32'

FLOAT32RAW = 'float32raw'

FLOAT64 = 'float64'

FLOAT64RAW = 'float64raw'

INT16 = 'int16'

INT16RAW = 'int16raw'

INT32 = 'int32'

INT32RAW = 'int32raw'

INT8 = 'int8'

INT8RAW = 'int8raw'

UINT16 = 'uint16'

UINT16RAW = 'uint16raw'

UINT8 = 'uint8'

UINT8RAW = 'uint8raw'

class geopyspark.geotrellis.constants.**ColorRamp**

ColorRamp names.

BLUE_TO_ORANGE = 'BlueToOrange'

BLUE_TO_RED = 'BlueToRed'

CLASSIFICATION_BOLD_LAND_USE = 'ClassificationBoldLandUse'

CLASSIFICATION_MUTED_TERRAIN = 'ClassificationMutedTerrain'

COOLWARM = 'CoolWarm'

GREEN_TO_RED_ORANGE = 'GreenToRedOrange'

HEATMAP_BLUE_TO_YELLOW_TO_RED_SPECTRUM = 'HeatmapBlueToYellowToRedSpectrum'

HEATMAP_DARK_RED_TO_YELLOW_WHITE = 'HeatmapDarkRedToYellowWhite'

HEATMAP_LIGHT_PURPLE_TO_DARK_PURPLE_TO_WHITE = 'HeatmapLightPurpleToDarkPurpleToWhite'

HEATMAP_YELLOW_TO_RED = 'HeatmapYellowToRed'

Hot = 'Hot'

INFERNO = 'Inferno'

LIGHT_TO_DARK_GREEN = 'LightToDarkGreen'

LIGHT_TO_DARK_SUNSET = 'LightToDarkSunset'

LIGHT_YELLOW_TO_ORANGE = 'LightYellowToOrange'

MAGMA = 'Magma'

PLASMA = 'Plasma'

VIRIDIS = 'Viridis'

geopyspark.geotrellis.constants.**DEFAULT_MAX_TILE_SIZE = 256**
The default byte size of each partition.

geopyspark.geotrellis.constants.**DEFAULT_PARTITION_BYTES = 1343225856**
The default number of bytes that should be read in at a time.

geopyspark.geotrellis.constants.**DEFAULT_CHUNK_SIZE = 65536**
The default name of the GeoTiff tag that contains the timestamp for the tile.

geopyspark.geotrellis.constants.**DEFAULT_GEOTIFF_TIME_TAG = 'TIFFTAG_DATETIME'**
The default pattern that will be parsed from the timeTag.

geopyspark.geotrellis.constants.**DEFAULT_GEOTIFF_TIME_FORMAT = 'yyyy:MM:dd HH:mm:ss'**
The default S3 Client to use when reading layers in.

class geopyspark.geotrellis.constants.**StorageMethod**
Internal storage methods for GeoTiffs.

STRIPED = 'Striped'

TILED = 'Tiled'

class geopyspark.geotrellis.constants.**ColorSpace**
Color space types for GeoTiffs.

BLACK_IS_ZERO = 1

CFA = 32803

CIE_LAB = 8

CMYK = 5

ICC_LAB = 9

ITU_LAB = 10

LINEAR_RAW = 34892

LOG_L = 32844

LOG_LUV = 32845

PALETTE = 3

RGB = 2

TRANSPARENCY_MASK = 4

WHITE_IS_ZERO = 0

Y_CB_CR = 6

class geopyspark.geotrellis.constants.**Compression**
Compression methods for GeoTiffs.

DEFLATE_COMPRESSION = 'DeflateCompression'

NO_COMPRESSION = 'NoCompression'

class geopyspark.geotrellis.constants.**Unit**
Represents the units of elevation.

FEET = 'Feet'

METERS = 'Meters'

2.13.5 geopyspark.geotrellis.cost_distance module

`geopyspark.geotrellis.cost_distance.`**`cost_distance`**(*friction_layer*, *geometries*, *max_distance*)

Performs cost distance of a TileLayer.

Parameters

- **`friction_layer`** (*`TiledRasterLayer`*) – `TiledRasterLayer` of a friction surface to traverse.
- **`geometries`** (*`list`*) – A list of shapely geometries to be used as a starting point.

  Note: All geometries must be in the same CRS as the TileLayer.

- **`max_distance`** (*`int or float`*) – The maximum cost that a path may reach before the operation. stops. This value can be an `int` or `float`.

Returns *`TiledRasterLayer`*

2.13.6 geopyspark.geotrellis.euclidean_distance module

`geopyspark.geotrellis.euclidean_distance.`**`euclidean_distance`**(*geometry*, *source_crs*, *zoom*, *cell_type=<CellType.FLOAT64: 'float64'>*)

Calculates the Euclidean distance of a Shapely geometry.

Parameters

- **`geometry`** (*`shapely.geometry`*) – The input geometry to compute the Euclidean distance for.
- **`source_crs`** (*`str or int`*) – The CRS of the input geometry.
- **`zoom`** (*`int`*) – The zoom level of the output raster.
- **`cell_type`** (str or *`CellType`*, optional) – The data type of the cells for the new layer. If not specified, then `CellType.FLOAT64` is used.

Note: This function may run very slowly for polygonal inputs if they cover many cells of the output raster.

Returns `TiledRasterLayer`

2.13.7 geopyspark.geotrellis.geotiff module

This module contains functions that create `RasterLayer` from files.

`geopyspark.geotrellis.geotiff.`**`get`**(*layer_type*, *uri*, *crs=None*, *max_tile_size=256*, *num_partitions=None*, *chunk_size=65536*, *partition_bytes=1343225856*, *time_tag='TIFFTAG_DATETIME'*, *time_format='yyyy:MM:dd HH:mm:ss'*, *delimiter=None*, *s3_client='default'*)

Creates a `RasterLayer` from GeoTiffs that are located on the local file system, `HDFS`, or `S3`.

Parameters

- **layer_type** (str or *LayerType*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.

  Note: All of the GeoTiffs must have the same saptial type.

- **uri** (*str or [str]*) – The path or list of paths to the desired tile(s)/directory(ies).
- **crs** (*str or int, optional*) – The CRS that the output tiles should be in. If `None`, then the CRS that the tiles were originally in will be used.
- **max_tile_size** (*int or None, optional*) – The max size of each tile in the resulting Layer. If the size is smaller than the read in tile, then that tile will be broken into smaller sections of the given size. Defaults to *DEFAULT_MAX_TILE_SIZE*. If `None`, then the whole tile will be read in.
- **num_partitions** (*int, optional*) – The number of partitions Spark will make when the data is repartitioned. If `None`, then the data will not be repartitioned.

  Note: If `max_tile_size` is also specified then this parameter will be ignored.

- **partition_bytes** (*int, optional*) – The desired number of bytes per partition. This is will ensure that at least one item is assigned for each partition. Defaults to *DEFAULT_PARTITION_BYTES*.
- **chunk_size** (*int, optional*) – How many bytes of the file should be read in at a time. Defaults to *DEFAULT_CHUNK_SIZE*.
- **time_tag** (*str, optional*) – The name of the tiff tag that contains the time stamp for the tile. Defaults to *DEFAULT_GEOTIFF_TIME_TAG*.
- **time_format** (*str, optional*) – The pattern of the time stamp to be parsed. Defaults to *DEFAULT_GEOTIFF_TIME_FORMAT*.
- **delimiter** (*str, optional*) – The delimiter to use for S3 object listings.

  Note: This parameter will only be used when reading from S3.

- **s3_client** (*str, optional*) – Which `S3Cleint` to use when reading GeoTiffs from S3. There are currently two options: `default` and `mock`. Defaults to `DEFAULT_S3_CLIENT`.

  Note: `mock` should only be used in unit tests and debugging.

Returns *RasterLayer*

2.13.8 geopyspark.geotrellis.hillshade module

`geopyspark.geotrellis.hillshade.`**`hillshade`**(*tiled_raster_layer*, *zfactor_calculator*, *band=0*, *azimuth=315.0*, *altitude=45.0*)

Computes Hillshade (shaded relief) from a raster.

The resulting raster will be a shaded relief map (a hill shading) based on the sun altitude, azimuth, and the `zfactor`. The `zfactor` is a conversion factor from map units to elevation units.

The `hillshade`` operation will be carried out in a `SQUARE` neighborhood with with an `extent` of 1. The `zfactor` will be derived from the `zfactor_calculator` for each `Tile` in the Layer. The resulting Layer will have a `cell_type` of `INT16` regardless of the input Layer's `cell_type`; as well as have a single band, that represents the calculated `hillshade`.

Returns a raster of ShortConstantNoDataCellType.

For descriptions of parameters, please see Esri Desktop's description of Hillshade.

Parameters

- **`tiled_raster_layer`** (*`TiledRasterLayer`*) – The base layer that contains the rasters used to compute the hillshade.
- **`zfactor_calculator`** (*`py4j.JavaObject`*) – A `JavaObject` that represents the Scala `ZFactorCalculator` class. This can be created using either the `zfactor_lat_lng_calculator()` or the `zfactor_calculator()` methods.
- **`band`** (*`int, optional`*) – The band of the raster to base the hillshade calculation on. Default is 0.
- **`azimuth`** (*`float, optional`*) – The azimuth angle of the source of light. Default value is 315.0.
- **`altitude`** (*`float, optional`*) – The angle of the altitude of the light above the horizon. Default is 45.0.

Returns *`TiledRasterLayer`*

2.13.9 geopyspark.geotrellis.histogram module

This module contains the `Histogram` class which is a wrapper of the GeoTrellis Histogram class.

class `geopyspark.geotrellis.histogram.`**`Histogram`**(*scala_histogram*)

A wrapper class for a GeoTrellis Histogram.

The underlying histogram is produced from the values within a *`TiledRasterLayer`*. These values represented by the histogram can either be `Int` or `Float` depending on the data type of the cells in the layer.

Parameters **`scala_histogram`** (*`py4j.JavaObject`*) – An instance of the GeoTrellis histogram.

`scala_histogram`

py4j.JavaObject – An instance of the GeoTrellis histogram.

`bin_counts`()

Returns a list of tuples where the key is the bin label value and the value is the label's respective count.

Returns [(int, int)] or [(float, int)]

`bucket_count`()

Returns the number of buckets within the histogram.

Returns int

`cdf`()

Returns the cdf of the distribution of the histogram.

Returns [(float, float)]

classmethod from_dict(*value*)
Encodes histogram as a dictionary

item_count(*item*)
Returns the total number of times a given item appears in the histogram.

Parameters **item** (*int or float*) – The value whose occurences should be counted.

Returns The total count of the occurences of `item` in the histogram.

Return type int

max()
The largest value of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns int or float

mean()
Determines the mean of the histogram.

Returns float

median()
Determines the median of the histogram.

Returns float

merge(*other_histogram*)
Merges this instance of `Histogram` with another. The resulting `Histogram` will contain values from both ‘‘Histogram‘‘s

Parameters **other_histogram** (*Histogram*) – The `Histogram` that should be merged with this instance.

Returns *Histogram*

min()
The smallest value of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns int or float

min_max()
The largest and smallest values of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns (int, int) or (float, float)

mode()
Determines the mode of the histogram.

This will return either an `int` or `float` depedning on the type of values within the histogram.

Returns int or float

quantile_breaks(*num_breaks*)
Returns quantile breaks for this Layer.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns [int]

to_dict()

Encodes histogram as a dictionary

Returns dict

values()

Lists each indiviual value within the histogram.

This will return a list of either ``int``s or ``float``s depedning on the type of values within the histogram.

Returns [int] or [float]

2.13.10 geopyspark.geotrellis.layer module

This module contains the RasterLayer and the TiledRasterLayer classes. Both of these classes are wrappers of their Scala counterparts. These will be used in leau of actual PySpark RDDs when performing operations.

class geopyspark.geotrellis.layer.**RasterLayer**(*layer_type*, *srdd*)

A wrapper of a RDD that contains GeoTrellis rasters.

Represents a layer that wraps a RDD that contains (K, V). Where K is either *ProjectedExtent* or *TemporalProjectedExtent* depending on the layer_type of the RDD, and V being a *Tile*.

The data held within this layer has not been tiled. Meaning the data has yet to be modified to fit a certain layout. See raster_rdd for more information.

Parameters

- **layer_type** (str or *LayerType*) – What the layer type of the geotiffs are. This is represented by either constants within LayerType or by a string.
- **srdd** (*py4j.java_gateway.JavaObject*) – The coresponding Scala class. This is what allows RasterLayer to access the various Scala methods.

pysc

pyspark.SparkContext – The SparkContext being used this session.

layer_type

LayerType – What the layer type of the geotiffs are.

srdd

py4j.java_gateway.JavaObject – The coresponding Scala class. This is what allows RasterLayer to access the various Scala methods.

bands(*band*)

Select a subsection of bands from the Tiles within the layer.

Note: There could be potential high performance cost if operations are performed between two sub-bands of a large data set.

Note: Due to the natue of GeoPySpark's backend, if selecting a band that is out of bounds then the error returned will be a py4j.protocol.Py4JJavaError and not a normal Python error.

Parameters band (*int or tuple or list or range*) – The band(s) to be selected from the Tiles. Can either be a single int, or a collection of ints.

Returns *RasterLayer* with the selected bands.

cache()

Persist this RDD with the default storage level (C{MEMORY_ONLY}).

collect_keys()

Returns a list of all of the keys in the layer.

Note: This method should only be called on layers with a smaller number of keys, as a large number could cause memory issues.

Returns `[:class:`~geopyspark.geotrellis.SpatialKey`]` or `[:ob:`~geopyspark.geotrellis.SpaceTimeKey`]`

collect_metadata(*layout=LocalLayout(tile_cols=256, tile_rows=256)*)

Iterate over the RDD records and generates layer metadata desribing the contained rasters.

:param layout (*LayoutDefinition* or: *GlobalLayout* or

***LocalLayout*, optional):** Target raster layout for the tiling operation.

Returns *Metadata*

convert_data_type(*new_type, no_data_value=None*)

Converts the underlying, raster values to a new `CellType`.

Parameters

- **new_type** (str or *CellType*) – The data type the cells should be to converted to.
- **no_data_value** (*int or float, optional*) – The value that should be marked as NoData.

Returns *RasterLayer*

Raises

- `ValueError` – If `no_data_value` is set and the `new_type` contains raw values.
- `ValueError` – If `no_data_value` is set and `new_type` is a boolean.

count()

Returns how many elements are within the wrapped RDD.

Returns The number of elements in the RDD.

Return type Int

filter_by_times(*time_intervals*)

Filters a `SPACETIME` layer by keeping only the values whose keys fall within a the given time interval(s).

Parameters **time_intervals** (`[datetime.datetime]`) – A list of the time intervals to query. This list can have one or multiple elements. If just a single element, then only exact matches with that given time will be kept. If there are multiple times given, then they are each paired together so that they form ranges of time. In the case where there are an odd number of elements, then the remaining time will be treated as a single query and not a range.

Note: If nothing intersects the given `time_intervals`, then the returned `RasterLayer` will be empty.

Returns `RasterLayer`

classmethod from_numpy_rdd(*layer_type*, *numpy_rdd*)

Create a `RasterLayer` from a numpy RDD.

Parameters

- **layer_type** (str or `LayerType`) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **numpy_rdd** (*pyspark.RDD*) – A PySpark RDD that contains tuples of either `ProjectedExtent`s or `TemporalProjectedExtent`s and rasters that are represented by a numpy array.

Returns `RasterLayer`

getNumPartitions()

Returns the number of partitions set for the wrapped RDD.

Returns The number of partitions.

Return type Int

get_class_histogram()

Creates a `Histogram` of integer values. Suitable for classification rasters with limited number values. If only single band is present histogram is returned directly.

Returns `Histogram` or [`Histogram`]

get_histogram()

Creates a `Histogram` for each band in the layer. If only single band is present histogram is returned directly.

Returns `Histogram` or [`Histogram`]

get_min_max()

Returns the maximum and minimum values of all of the rasters in the layer.

Returns (float, float)

get_partition_strategy()

Returns the partitioning strategy if the layer has one.

Returns `HashPartitioner` or `SpatialPartitioner` or `SpaceTimePartitionStrategy` or `None`

get_quantile_breaks(*num_breaks*)

Returns quantile breaks for this Layer.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns `[float]`

get_quantile_breaks_exact_int(*num_breaks*)

Returns quantile breaks for this Layer. This version uses the `FastMapHistogram`, which counts exact integer values. If your layer has too many values, this can cause memory errors.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns `[int]`

isEmpty()

Returns a bool that is True if the layer is empty and False if it is not.

Returns Are there elements within the layer

Return type bool

`map_cells`(*func*)

Maps over the cells of each `Tile` within the layer with a given function.

Note: This operation first needs to deserialize the wrapped `RDD` into Python and then serialize the `RDD` back into a `TiledRasterRDD` once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **`func`** (*`cells, nd => cells`*) – A function that takes two arguements: `cells` and `nd`. Where `cells` is the numpy array and `nd` is the `no_data_value` of the `Tile`. It returns `cells` which are the new cells values of the `Tile` represented as a numpy array.

Returns *`RasterLayer`*

`map_tiles`(*func*)

Maps over each `Tile` within the layer with a given function.

Note: This operation first needs to deserialize the wrapped `RDD` into Python and then serialize the `RDD` back into a `RasterRDD` once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **`func`** (*`Tile`* => *`Tile`*) – A function that takes a `Tile` and returns a `Tile`.

Returns *`RasterLayer`*

`merge`(*partition_strategy=None*)

Merges the `Tile` of each `K` together to produce a single `Tile`.

This method will reduce each value by its key within the layer to produce a single `(K, V)` for every `K`. In order to achieve this, each `Tile` that shares a `K` is merged together to form a single `Tile`. This is done by replacing one `Tile`'s cells with another's. Not all cells, if any, may be replaced, however. The following steps are taken to determine if a cell's value should be replaced:

1. If the cell contains a `NoData` value, then it will be replaced.
2. If no `NoData` value is set, then a cell with a value of 0 will be replaced.
3. If neither of the above are true, then the cell retain its value.

Parameters

- **`num_partitions`** (*`int, optional`*) – The number of partitions that the resulting layer should be partitioned with. If `None`, then the `num_partitions` will the number of partitions the layer curretly has.
- **`partition_strategy`** (*`HashPartitionStrategy`* or `SpatialPartitioinStrategy` or *`SpaceTimePartitionStrategy`*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns `RasterLayer`

`partitionBy`(*partition_strategy=None*)

Repartitions the layer using the given partitioning strategy.

Parameters **`partition_strategy`** (`HashPartitionStrategy` or `SpatialPartitioinStrategy` or `SpaceTimePartitionStrategy`, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

If `None`, then the output layer will be the same as the source layer.

If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns `RasterLayer`

`persist`(*storageLevel=StorageLevel(False, True, False, False, 1)*)

Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (C{MEMORY_ONLY}).

`reclassify`(*value_map, data_type, classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>, replace_nodata_with=None, fallback_value=None, strict=False*)

Changes the cell values of a raster based on how the data is broken up in the given `value_map`.

Parameters

- **`value_map`** (*`dict`*) – A `dict` whose keys represent values where a break should occur and its values are the new value the cells within the break should become.
- **`data_type`** (*`type`*) – The type of the values within the rasters. Can either be int or float.
- **`classification_strategy`** (str or `ClassificationStrategy`, optional) – How the cells should be classified along the breaks. If unspecified, then `ClassificationStrategy.LESS_THAN_OR_EQUAL_TO` will be used.
- **`replace_nodata_with`** (*`int or float, optional`*) – When remapping values, `NoData` values must be treated separately. If `NoData` values are intended to be replaced during the reclassify, this variable should be set to the intended value. If unspecified, `NoData` values will be preserved.

  Note: Specifying `replace_nodata_with` will change the value of given cells, but the `NoData` value of the layer will remain unchanged.

- **`fallback_value`** (*`int or float, optional`*) – Represents the value that should be used when a cell's value does not fall within the `classification_strategy`. Default is to use the layer's `NoData` value.

- **strict** (*bool, optional*) – Determines whether or not an error should be thrown if a cell's value does not fall within the `classification_strategy`. Default is, `False`.

Returns *RasterLayer*

repartition(*num_partitions=None*)

Repartitions the layer to have a different number of partitions.

Parameters num_partitions (*int, optional*) – Desired number of partitions. Default is, `None` .If `None`, then the exisiting number of partitions will be used.

Returns *RasterLayer*

reproject(*target_crs*, *resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'Nearest-Neighbor'>*)

Reproject rasters to `target_crs`. The reproject does not sample past tile boundary.

Parameters

- **target_crs** (*str or int*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string.
- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use for the reprojection. If none is specified, then `ResampleMethods.NEAREST_NEIGHBOR` is used.

Returns *RasterLayer*

tile_to_layout(*layout=LocalLayout(tile_cols=256, tile_rows=256)*, *target_crs=None*, *resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*, *partition_strategy=None*)

Cut tiles to layout and merge overlapping tiles. This will produce unique keys.

Parameters

- **layout** (*Metadata* or `TiledRasterLayer` or *LayoutDefinition* or *GlobalLayout* or *LocalLayout*) – Target raster layout for the tiling operation.
- **target_crs** (*str or int, optional*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string. If `None`, no reproject will be perfomed.
- **resample_method** (str or *ResampleMethod*, optional) – The cell resample method to used during the tiling operation. Default is``ResampleMethods.NEAREST_NEIGHBOR``.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

`to_geotiff_rdd`(*storage_method=<StorageMethod.STRIPED: 'Striped'>, rows_per_strip=None, tile_dimensions=(256, 256), compression=<Compression.NO_COMPRESSION: 'NoCompression'>, color_space=<ColorSpace.BLACK_IS_ZERO: 1>, color_map=None, head_tags=None, band_tags=None*)

Converts the rasters within this layer to GeoTiffs which are then converted to bytes. This is returned as a `RDD[(K, bytes)]`. Where `K` is either `ProjectedExtent` or `TemporalProjectedExtent`.

Parameters

- **`storage_method`** (str or *`StorageMethod`*, optional) – How the segments within the GeoTiffs should be arranged. Default is `StorageMethod.STRIPED`.
- **`rows_per_strip`** (*`int, optional`*) – How many rows should be in each strip segment of the GeoTiffs if `storage_method` is `StorageMethod.STRIPED`. If `None`, then the strip size will default to a value that is 8K or less.
- **`tile_dimensions`** (*`(int, int), optional`*) – The length and width for each tile segment of the GeoTiff if `storage_method` is `StorageMethod.TILED`. If `None` then the default size is `(256, 256)`.
- **`compression`** (str or *`Compression`*, optional) – How the data should be compressed. Defaults to `Compression.NO_COMPRESSION`.
- **`color_space`** (str or *`ColorSpace`*, optional) – How the colors should be organized in the GeoTiffs. Defaults to `ColorSpace.BLACK_IS_ZERO`.
- **`color_map`** (*`ColorMap`*, optional) – A `ColorMap` instance used to color the GeoTiffs to a different gradient.
- **`head_tags`** (*`dict, optional`*) – A `dict` where each key and value is a `str`.
- **`band_tags`** (*`list, optional`*) – A `list` of `dict`s where each key and value is a `str`.
- **`Note`** – For more information on the contents of the tags, see www.gdal.org/gdal_datamodel.html

Returns RDD[(K, bytes)]

`to_numpy_rdd`()

Converts a `RasterLayer` to a numpy RDD.

Note: Depending on the size of the data stored within the RDD, this can be an exspensive operation and should be used with caution.

Returns RDD

`to_png_rdd`(*color_map*)

Converts the rasters within this layer to PNGs which are then converted to bytes. This is returned as a RDD[(K, bytes)].

Parameters **`color_map`** (*`ColorMap`*) – A `ColorMap` instance used to color the PNGs.

Returns RDD[(K, bytes)]

`to_spatial_layer`(*target_time=None*)

Converts a `RasterLayer` with a `layout_type` of `LayoutType.SPACETIME` to a `RasterLayer` with a `layout_type` of `LayoutType.SPATIAL`.

Parameters **`target_time`** (`datetime.datetime`, optional) – The instance of interest. If set, the resulting `RasterLayer` will only contain keys that contained the given instance. If `None`, then all values within the layer will be kept.

Returns *`RasterLayer`*

Raises `ValueError` – If the layer already has a `layout_type` of `LayoutType.SPATIAL`.

`unpersist``()`

Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

`with_no_data``(`*no_data_value*`)`

Changes the `NoData` value of the layer with the new given value.

It is possible to specify a `NoData` value for layers with raw values. The resulting layer will be of the same `CellType` but with a user defined `NoData` value. For example, if a layer has a `CellType` of `float32raw` and a `no_data_value` of `-10` is given, then the produced layer will have a `CellType` of `float32ud-10.0`.

If the target layer has a `bool CellType`, then the `no_data_value` will be ignored and the result layer will be the same as the origin. In order to assign a `NoData` value to a `bool` layer, the *`convert_data_type()`* method must be used.

Parameters **`no_data_value`** (*`int or float`*) – The new `NoData` value of the layer.

Returns *`RasterLayer`*

`wrapped_rdds``()`

Returns the list of RDD-containing objects wrapped by this object. The default implementation assumes that subclass contains a single RDD container, srdd, which implements the persist() and unpersist() methods.

class `geopyspark.geotrellis.layer.`**`TiledRasterLayer`**`(`*layer_type*, *srdd*`)`

Wraps a RDD of tiled, GeoTrellis rasters.

Represents a RDD that contains `(K, V)`. Where `K` is either *`SpatialKey`* or *`SpaceTimeKey`* depending on the `layer_type` of the RDD, and `V` being a *`Tile`*.

The data held within the layer is tiled. This means that the rasters have been modified to fit a larger layout. For more information, see tiled-raster-rdd.

Parameters

- **`layer_type`** (str or *`LayerType`*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **`srdd`** (*`py4j.java_gateway.JavaObject`*) – The coresponding Scala class. This is what allows `TiledRasterLayer` to access the various Scala methods.

`pysc`

pyspark.SparkContext – The `SparkContext` being used this session.

`layer_type`

`LayerType` – What the layer type of the geotiffs are.

`srdd`

py4j.java_gateway.JavaObject – The coresponding Scala class. This is what allows `RasterLayer` to access the various Scala methods.

`is_floating_point_layer`

bool – Whether the data within the `TiledRasterLayer` is floating point or not.

`layer_metadata`
`Metadata` – The layer metadata associated with this layer.

`zoom_level`
int – The zoom level of the layer. Can be `None`.

`aggregate_by_cell`(*operation*)
Computes an aggregate summary for each cell of all of the values for each key.

The `operation` given is a local map algebra function that will be applied to all values that share the same key. If there are multiple copies of the same key in the layer, then this method will reduce all instances of the `(K, Tile)` pairs into a single element. This resulting `(K, Tile)`'s `Tile` will contain the aggregate summaries of each cell of the reduced `Tile`s that had the same `K`.

Note: Not all `Operations` are supported. Only `SUM`, `MIN`, `MAX`, `MEAN`, `VARIANCE`, AND `STANDARD_DEVIATION` can be used.

Note: If calculating `VARIANCE` or `STANDARD_DEVIATION`, then any `K` that is a single copy will have a resulting `Tile` that is filled with `NoData` values. This is because the variance of a single element is undefined.

Parameters **`operation`** (str or *`Operation`*) – The aggregate operation to be performed.

Returns *`TiledRasterLayer`*

`bands`(*band*)
Select a subsection of bands from the `Tile`s within the layer.

Note: There could be potential high performance cost if operations are performed between two sub-bands of a large data set.

Note: Due to the natue of GeoPySpark's backend, if selecting a band that is out of bounds then the error returned will be a `py4j.protocol.Py4JJavaError` and not a normal Python error.

Parameters **`band`** (*`int or tuple or list or range`*) – The band(s) to be selected from the `Tile`s. Can either be a single int, or a collection of ints.

Returns *`TiledRasterLayer`* with the selected bands.

`cache`()
Persist this RDD with the default storage level (C{MEMORY_ONLY}).

`collect_keys`()
Returns a list of all of the keys in the layer.

Note: This method should only be called on layers with a smaller number of keys, as a large number could cause memory issues.

Returns [:class:`~geopyspark.geotrellis.ProjectedExtent`] or [:class:`~geopyspark.geotrellis.TemporalProjectedExtent`]

convert_data_type(*new_type*, *no_data_value=None*)

Converts the underlying, raster values to a new `CellType`.

Parameters

- **new_type** (str or *CellType*) – The data type the cells should be to converted to.
- **no_data_value** (*int or float, optional*) – The value that should be marked as NoData.

Returns *TiledRasterLayer*

Raises

- `ValueError` – If `no_data_value` is set and the `new_type` contains raw values.
- `ValueError` – If `no_data_value` is set and `new_type` is a boolean.

count()

Returns how many elements are within the wrapped RDD.

Returns The number of elements in the RDD.

Return type Int

filter_by_times(*time_intervals*)

Filters a `SPACETIME` layer by keeping only the values whose keys fall within a the given time interval(s).

Parameters **time_intervals** ([datetime.datetime]) – A list of the time intervals to query. This list can have one or multiple elements. If just a single element, then only exact matches with that given time will be kept. If there are multiple times given, then they are each paired together so that they form ranges of time. In the case where there are an odd number of elements, then the remaining time will be treated as a single query and not a range.

Note: If nothing intersects the given `time_intervals`, then the returned `TiledRasterLayer` will be empty.

Returns *TiledRasterLayer*

focal(*operation*, *neighborhood=None*, *param_1=None*, *param_2=None*, *param_3=None*, *partition_strategy=None*)

Performs the given focal operation on the layers contained in the Layer.

Parameters

- **operation** (str or *Operation*) – The focal operation to be performed.
- **neighborhood** (str or `Neighborhood`, optional) – The type of neighborhood to use in the focal operation. This can be represented by either an instance of `Neighborhood`, or by a constant.
- **param_1** (*int or float, optional*) – The first argument of `neighborhood`.
- **param_2** (*int or float, optional*) – The second argument of the `neighborhood`.
- **param_3** (*int or float, optional*) – The third argument of the `neighborhood`.

- **partition_strategy** (*HashPartitionStrategy* or SpatialPartitioinStrategy or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Note: `param` only need to be set if `neighborhood` is not an instance of `Neighborhood` or if `neighborhood` is `None`.

Any `param` that is not set will default to 0.0.

If `neighborhood` is `None` then `operation` **must** be `Operation.ASPECT`.

Returns *TiledRasterLayer*

Raises

- `ValueError` – If `operation` is not a known operation.
- `ValueError` – If `neighborhood` is not a known neighborhood.
- `ValueError` – If `neighborhood` was not set, and `operation` is not `Operation.ASPECT`.

classmethod from_numpy_rdd(*layer_type*, *numpy_rdd*, *metadata*, *zoom_level=None*)

Create a `TiledRasterLayer` from a numpy RDD.

Parameters

- **layer_type** (str or *LayerType*) – What the layer type of the geotiffs are. This is represented by either constants within `LayerType` or by a string.
- **numpy_rdd** (*pyspark.RDD*) – A PySpark RDD that contains tuples of either *SpatialKey* or *SpaceTimeKey* and rasters that are represented by a numpy array.
- **metadata** (*Metadata*) – The `Metadata` of the `TiledRasterLayer` instance.
- **zoom_level** (*int, optional*) – The `zoom_level` the resulting *TiledRasterLayer* should have. If `None`, then the returned layer's `zoom_level` will be `None`.

Returns *TiledRasterLayer*

getNumPartitions()

Returns the number of partitions set for the wrapped RDD.

Returns The number of partitions.

Return type Int

get_class_histogram()

Creates a `Histogram` of integer values. Suitable for classification rasters with limited number values. If only single band is present histogram is returned directly.

Returns *Histogram* or [*Histogram*]

get_histogram()

Creates a `Histogram` for each band in the layer. If only single band is present histogram is returned directly.

Returns *Histogram* or [*Histogram*]

get_min_max()

Returns the maximum and minimum values of all of the rasters in the layer.

Returns (float, float)

get_partition_strategy()

Returns the partitioning strategy if the layer has one.

Returns `HashPartitioner` or `SpatialPartitioner` or *SpaceTimePartitionStrategy* or `None`

get_point_values(*points*, *resample_method=None*)

Returns the values of the layer at given points.

Note: Only points that are contained within a layer will be sampled. This means that if a point lies on the southern or eastern boundary of a cell, it will not be sampled.

Parameters

- **or {k** (*points ([shapely.geometry.Point]*) – shapely.geometry.Point}): Either a list of, or a dictionary whose values are `shapely.geometry.Points`. If a dictionary, then the type of its keys does not matter. These points must be in the same projection as the tiles within the layer.
- **resample_method** (str or `ResampleMethod`, optional) – The resampling method to use before obtaining the point values. If not specified, then `None` is used.

Note: Not all `ResampleMethods` can be used to resample point values. `ResampleMethod.NEAREST_NEIGHBOR`, `ResampleMethod.BILINEAR``, `ResampleMethod.CUBIC_CONVOLUTION`, and `ResampleMethod.CUBIC_SPLINE` are the only ones that can be used.

Returns

The return type will vary depending on the type of `points` and the `layer_type` of the sampled layer.

If `points` is a `list` and the `layer_type` is `SPATIAL`: [(shapely.geometry.Point, [float])]

If `points` is a `list` and the `layer_type` is `SPACETIME`: [(shapely.geometry.Point, [(datetime.datetime, [float])])]

If `points` is a `dict` and the `layer_type` is `SPATIAL`: {k: (shapely.geometry.Point, [float])}

If `points` is a `dict` and the `layer_type` is `SPACETIME`: {k: (shapely.geometry.Point, [(datetime.datetime, [float])])}

The `shapely.geometry.Point` in all of these returns is the original sampled point given. The `[float]` are the sampled values, one for each band. If the `layer_type` was `SPACETIME`, then the timestamp will also be included in the results represented by a `datetime.datetime` instance. These times and their associated values will be given as a list of tuples for each point.

Note: The sampled values will always be returned as `floats`. Regardless of the `cellType` of the layer.

If `points` was given as a `dict` then the keys of that dictionary will be the keys in the returned `dict`.

`get_quantile_breaks`(*num_breaks*)
Returns quantile breaks for this Layer.

Parameters **`num_breaks`** (*`int`*) – The number of breaks to return.

Returns `[float]`

`get_quantile_breaks_exact_int`(*num_breaks*)
Returns quantile breaks for this Layer. This version uses the `FastMapHistogram`, which counts exact integer values. If your layer has too many values, this can cause memory errors.

Parameters **`num_breaks`** (*`int`*) – The number of breaks to return.

Returns `[int]`

`isEmpty`()
Returns a bool that is True if the layer is empty and False if it is not.

Returns Are there elements within the layer

Return type bool

`local_max`(*value*)
Determines the maximum value for each cell of each `Tile` in the layer.

This method takes a `max_constant` that is compared to each cell in the layer. If `max_constant` is larger, then the resulting cell value will be that value. Otherwise, that cell will retain its original value.

Note: `NoData` values are handled such that taking the max between a normal value and `NoData` value will always result in `NoData`.

Parameters **`value`** (int or float or *`TiledRasterLayer`*) – The constant value that will be compared to each cell. If this is a `TiledRasterLayer`, then `Tiles` who share a key will have each of their cell values compared.

Returns *`TiledRasterLayer`*

`lookup`(*col*, *row*)
Return the value(s) in the image of a particular `SpatialKey` (given by col and row).

Parameters

- **`col`** (*`int`*) – The `SpatialKey` column.
- **`row`** (*`int`*) – The `SpatialKey` row.

Returns [*`Tile`*]

Raises

- `ValueError` – If using lookup on a non `LayerType.SPATIAL` `TiledRasterLayer`.
- `IndexError` – If col and row are not within the `TiledRasterLayer`'s bounds.

map_cells (*func*)

Maps over the cells of each `Tile` within the layer with a given function.

Note: This operation first needs to deserialize the wrapped `RDD` into Python and then serialize the `RDD` back into a `TiledRasterRDD` once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **func** (*cells, nd => cells*) – A function that takes two arguements: `cells` and `nd`. Where `cells` is the numpy array and `nd` is the `no_data_value` of the tile. It returns `cells` which are the new cells values of the tile represented as a numpy array.

Returns *TiledRasterLayer*

map_tiles (*func*)

Maps over each `Tile` within the layer with a given function.

Note: This operation first needs to deserialize the wrapped `RDD` into Python and then serialize the `RDD` back into a `TiledRasterRDD` once the mapping is done. Thus, it is advised to chain together operations to reduce performance cost.

Parameters **func** (*Tile* => *Tile*) – A function that takes a `Tile` and returns a `Tile`.

Returns *TiledRasterLayer*

mask (*geometries, partition_strategy=None, options=RasterizerOptions(includePartial=True, sampleType='PixelIsPoint')*)

Masks the `TiledRasterLayer` so that only values that intersect the geometries will be available.

Parameters

- **geometries** (*shapely.geometry or [shapely.geometry] or pyspark.RDD[shapely.geometry]*) – Either a single, list, or Python `RDD` of shapely geometry/ies to mask the layer.

  Note: All geometries must be in the same CRS as the TileLayer.

- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Note: This parameter will only be used if `geometries` is a `pyspark.RDD`.

- **options** (*RasterizerOptions*, optional) – During the mask operation, rasterization occurs. These options will change the pixel rasterization behavior. Default behavior is to include partial pixel intersection and to treat pixels as points.

Note: This parameter will only be used if `geometries` is a `pyspark.RDD`.

Returns *TiledRasterLayer*

merge (*partition_strategy=None*)

Merges the `Tile` of each `K` together to produce a single `Tile`.

This method will reduce each value by its key within the layer to produce a single `(K, V)` for every `K`. In order to achieve this, each `Tile` that shares a `K` is merged together to form a single `Tile`. This is done by replacing one `Tile`'s cells with another's. Not all cells, if any, may be replaced, however. The following steps are taken to determine if a cell's value should be replaced:

1. If the cell contains a `NoData` value, then it will be replaced.
2. If no `NoData` value is set, then a cell with a value of 0 will be replaced.
3. If neither of the above are true, then the cell retain its value.

Parameters

- **num_partitions** (*int, optional*) – The number of partitions that the resulting layer should be partitioned with. If `None`, then the `num_partitions` will the number of partitions the layer curretly has.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

normalize (*new_min, new_max, old_min=None, old_max=None*)

Finds the min value that is contained within the given geometry.

Note: If `old_max - old_min <= 0` or `new_max - new_min <= 0`, then the normalization will fail.

Parameters

- **old_min** (*int or float, optional*) – Old minimum. If not given, then the minimum value of this layer will be used.
- **old_max** (*int or float, optional*) – Old maximum. If not given, then the minimum value of this layer will be used.
- **new_min** (*int or float*) – New minimum to normalize to.
- **new_max** (*int or float*) – New maximum to normalize to.

Returns *TiledRasterLayer*

partitionBy (*partition_strategy=None*)
Repartitions the layer using the given partitioning strategy.

Parameters **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

If `None`, then the output layer will be the same as the source layer.

If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

persist (*storageLevel=StorageLevel(False, True, False, False, 1)*)
Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (C{MEMORY_ONLY}).

polygonal_max (*geometry, data_type*)
Finds the max value for each band that is contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKB representation of the geometry.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.

Returns [int] or [float] depending on `data_type`.

Raises `TypeError` – If `data_type` is not an int or float.

polygonal_mean (*geometry*)
Finds the mean of all of the values for each band that are contained within the given geometry.

Parameters **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKB representation of the geometry.

Returns [float]

polygonal_min (*geometry, data_type*)
Finds the min value for each band that is contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely Polygon or MultiPolygon that represents the area where the summary should be computed; or a WKB representation of the geometry.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.

Returns [int] or [float] depending on data_type.

Raises TypeError – If data_type is not an int or float.

polygonal_sum(*geometry*, *data_type*)

Finds the sum of all of the values in each band that are contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon or bytes*) – A Shapely Polygon or MultiPolygon that represents the area where the summary should be computed; or a WKB representation of the geometry.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.

Returns [int] or [float] depending on data_type.

Raises TypeError – If data_type is not an int or float.

pyramid(*resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*, *partition_strategy=None*)

Creates a layer Pyramid where the resolution is halved per level.

Parameters

- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use when building the pyramid. Default is ResampleMethods.NEAREST_NEIGHBOR.
- **partition_strategy** (*HashPartitionStrategy* or SpatialPartitioinStrategy or *SpaceTimePartitionStrategy*, optional) – Sets the Partitioner for the resulting layer and how many partitions it has. Default is, None.

  If None, then the output layer will be the same Partitioner and number of partitions as the source layer.

  If partition_strategy is set but has no num_partitions, then the resulting layer will have the Partioner specified in the strategy with the with same number of partitions the source layer had.

  If partition_strategy is set and has a num_partitions, then the resulting layer will have the Partioner and number of partitions specified in the strategy.

Returns *Pyramid*.

Raises ValueError – If this layer layout is not of GlobalLayout type.

reclassify(*value_map*, *data_type*, *classification_strategy=<ClassificationStrategy.LESS_THAN_OR_EQUAL_TO: 'LessThanOrEqualTo'>*, *replace_nodata_with=None*, *fallback_value=None*, *strict=False*)

Changes the cell values of a raster based on how the data is broken up in the given value_map.

Parameters

- **value_map** (*dict*) – A dict whose keys represent values where a break should occur and its values are the new value the cells within the break should become.
- **data_type** (*type*) – The type of the values within the rasters. Can either be int or float.
- **classification_strategy** (str or *ClassificationStrategy*, optional) – How the cells should be classified along the breaks. If unspecified, then ClassificationStrategy.LESS_THAN_OR_EQUAL_TO will be used.
- **replace_nodata_with** (*int or float, optional*) – When remapping values, NoData values must be treated separately. If NoData values are intended to be replaced during the reclassify, this variable should be set to the intended value. If unspecified, NoData values will be preserved.

  Note: Specifying replace_nodata_with will change the value of given cells, but the NoData value of the layer will remain unchanged.

- **fallback_value** (*int or float, optional*) – Represents the value that should be used when a cell's value does not fall within the classification_strategy. Default is to use the layer's NoData value.
- **strict** (*bool, optional*) – Determines whether or not an error should be thrown if a cell's value does not fall within the classification_strategy. Default is, False.

Returns *TiledRasterLayer*

repartition(*num_partitions=None*)
Repartitions the layer to have a different number of partitions.

Parameters num_partitions (*int, optional*) – Desired number of partitions. Default is, None .If None, then the exisiting number of partitions will be used.

Returns *TiledRasterLayer*

reproject(*target_crs*, *resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'Nearest-Neighbor'>*)
Reproject rasters to target_crs. The reproject does not sample past tile boundary.

Parameters

- **target_crs** (*str or int*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string.
- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use for the reprojection. If none is specified, then ResampleMethods.NEAREST_NEIGHBOR is used.

Returns *TiledRasterLayer*

save_stitched(*path*, *crop_bounds=None*, *crop_dimensions=None*)
Stitch all of the rasters within the Layer into one raster and then saves it to a given path.

Parameters

- **path** (*str*) – The path of the geotiff to save. The path must be on the local file system.
- **crop_bounds** (*Extent*, optional) – The sub Extent with which to crop the raster before saving. If None, then the whole raster will be saved.

- **crop_dimensions** (*tuple(int) or list(int), optional*) – cols and rows of the image to save represented as either a tuple or list. If `None` then all cols and rows of the raster will be save.

Note: This can only be used on `LayerType.SPATIAL` `TiledRasterLayers`.

Note: If `crop_dimensions` is set then `crop_bounds` must also be set.

slope(*zfactor_calculator*)

Performs the Slope, focal operation on the first band of each `Tile` in the Layer.

The Slope operation will be carried out in a `SQUARE` neighborhood with with an `extent` of 1. A `zfactor` will be derived from the `zfactor_calculator` for each `Tile` in the Layer. The resulting Layer will have a `cell_type` of `FLOAT64` regardless of the input Layer's `cell_type`; as well as have a single band, that represents the calculated slope.

Parameters **zfactor_calculator** (*py4j.JavaObject*) – A `JavaObject` that represents the Scala `ZFactorCalculator` class. This can be created using either the `zfactor_lat_lng_calculator()` or the `zfactor_calculator()` methods.

Returns *TiledRasterLayer*

stitch()

Stitch all of the rasters within the Layer into one raster.

Note: This can only be used on `LayerType.SPATIAL` `TiledRasterLayers`.

Returns *Tile*

tile_to_layout(*layout*, *target_crs=None*, *resample_method=<ResampleMethod.NEAREST_NEIGHBOR: 'NearestNeighbor'>*, *partition_strategy=None*)

Cut tiles to a given layout and merge overlapping tiles. This will produce unique keys.

Parameters

- **layout** (*LayoutDefinition* or *Metadata* or `TiledRasterLayer` or *GlobalLayout* or *LocalLayout*) – Target raster layout for the tiling operation.
- **target_crs** (*str or int, optional*) – Target CRS of reprojection. Either EPSG code, well-known name, or a PROJ.4 string. If `None`, no reproject will be perfomed.
- **resample_method** (str or *ResampleMethod*, optional) – The resample method to use for the reprojection. If none is specified, then `ResampleMethods.NEAREST_NEIGHBOR` is used.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy` or *SpaceTimePartitionStrategy*, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will be the same `Partitioner` and number of partitions as the source layer.

If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *`TiledRasterLayer`*

`to_geotiff_rdd`(*storage_method=<StorageMethod.STRIPED: 'Striped'>, rows_per_strip=None, tile_dimensions=(256, 256), compression=<Compression.NO_COMPRESSION: 'NoCompression'>, color_space=<ColorSpace.BLACK_IS_ZERO: 1>, color_map=None, head_tags=None, band_tags=None*)

Converts the rasters within this layer to GeoTiffs which are then converted to bytes. This is returned as a `RDD[(K, bytes)]`. Where `K` is either `SpatialKey` or `SpaceTimeKey`.

Parameters

- **`storage_method`** (str or *`StorageMethod`*, optional) – How the segments within the GeoTiffs should be arranged. Default is `StorageMethod.STRIPED`.
- **`rows_per_strip`** (*`int, optional`*) – How many rows should be in each strip segment of the GeoTiffs if `storage_method` is `StorageMethod.STRIPED`. If `None`, then the strip size will default to a value that is 8K or less.
- **`tile_dimensions`** (*`(int, int), optional`*) – The length and width for each tile segment of the GeoTiff if `storage_method` is `StorageMethod.TILED`. If `None` then the default size is `(256, 256)`.
- **`compression`** (str or *`Compression`*, optional) – How the data should be compressed. Defaults to `Compression.NO_COMPRESSION`.
- **`color_space`** (str or *`ColorSpace`*, optional) – How the colors should be organized in the GeoTiffs. Defaults to `ColorSpace.BLACK_IS_ZERO`.
- **`color_map`** (*`ColorMap`*, optional) – A `ColorMap` instance used to color the GeoTiffs to a different gradient.
- **`head_tags`** (*`dict, optional`*) – A `dict` where each key and value is a `str`.
- **`band_tags`** (*`list, optional`*) – A `list` of `dict`s where each key and value is a `str`.
- **`Note`** – For more information on the contents of the tags, see www.gdal.org/gdal_datamodel.html

Returns RDD[(K, bytes)]

`to_numpy_rdd`()

Converts a `TiledRasterLayer` to a numpy RDD.

Note: Depending on the size of the data stored within the RDD, this can be an exspensive operation and should be used with caution.

Returns RDD

`to_png_rdd`(*color_map*)

Converts the rasters within this layer to PNGs which are then converted to bytes. This is returned as a RDD[(K, bytes)].

Parameters **`color_map`** (*`ColorMap`*) – A `ColorMap` instance used to color the PNGs.

Returns RDD[(K, bytes)]

`to_spatial_layer`(*target_time=None*)

Converts a `TiledRasterLayer` with a `layout_type` of `LayoutType.SPACETIME` to a `TiledRasterLayer` with a `layout_type` of `LayoutType.SPATIAL`.

Parameters **`target_time`** (`datetime.datetime`, optional) – The instance of interest. If set, the resulting `TiledRasterLayer` will only contain keys that contained the given instance. If `None`, then all values within the layer will be kept.

Returns *`TiledRasterLayer`*

Raises `ValueError` – If the layer already has a `layout_type` of `LayoutType.SPATIAL`.

`tobler`()

Generates a Tobler walking speed layer from an elevation layer.

Note: This method has a known issue where the Tobler calculation is direction agnostic. Thus, all slopes are assumed to be uphill. This can result it incorrect results. A fix is currently being worked on.

Returns *`TiledRasterLayer`*

`unpersist`()

Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

`with_no_data`(*no_data_value*)

Changes the `NoData` value of the layer with the new given value.

It is possible to specify a `NoData` value for layers with raw values. The resulting layer will be of the same `CellType` but with a user defined `NoData` value. For example, if a layer has a `CellType` of `float32raw` and a `no_data_value` of `-10` is given, then the produced layer will have a `CellType` of `float32ud-10.0`.

If the target layer has a `bool` `CellType`, then the `no_data_value` will be ignored and the result layer will be the same as the origin. In order to assign a `NoData` value to a `bool` layer, the *`convert_data_type()`* method must be used.

Parameters **`no_data_value`** (*`int or float`*) – The new `NoData` value of the layer.

Returns *`TiledRasterLayer`*

`wrapped_rdds`()

Returns the list of RDD-containing objects wrapped by this object. The default implementation assumes that subclass contains a single RDD container, srdd, which implements the persist() and unpersist() methods.

class `geopyspark.geotrellis.layer.`**`Pyramid`**(*levels*)

Contains a list of `TiledRasterLayer`s that make up a tile pyramid. Each layer represents a level within the pyramid. This class is used when creating a tile server.

Map algebra can performed on instances of this class.

Parameters **`levels`** (*`list or dict`*) – A list of `TiledRasterLayers` or a dict of `TiledRasterLayers` where the value is the layer itself and the key is its given zoom level.

`pysc`

pyspark.SparkContext – The `SparkContext` being used this session.

layer_type (class
~*geopyspark.geotrellis.constants.LayerType*): What the layer type of the geotiffs are.

levels
dict – A dict of `TiledRasterLayers` where the value is the layer itself and the key is its given zoom level.

max_zoom
int – The highest zoom level of the pyramid.

is_cached
bool – Signals whether or not the internal RDDs are cached. Default is `False`.

histogram
`Histogram` – The `Histogram` that represents the layer with the max zoomw. Will not be calculated unless the *`get_histogram()`* method is used. Otherwise, its value is `None`.

Raises `TypeError` – If `levels` is neither a list or dict.

cache()
Persist this RDD with the default storage level (C{MEMORY_ONLY}).

count()
Returns how many elements are within the wrapped RDD.

Returns The number of elements in the RDD.

Return type Int

getNumPartitions()
Returns the number of partitions set for the wrapped RDD.

Returns The number of partitions.

Return type Int

get_histogram()
Calculates the `Histogram` for the layer with the max zoom.

Returns *`Histogram`*

get_partition_strategy()
Returns the partitioning strategy if the layer has one.

Returns `HashPartitioner` or `SpatialPartitioner` or *`SpaceTimePartitionStrategy`* or `None`

isEmpty()
Returns a bool that is True if the layer is empty and False if it is not.

Returns Are there elements within the layer

Return type bool

persist(*storageLevel=StorageLevel(False*, *True*, *False*, *False*, *1)*)
Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (C{MEMORY_ONLY}).

unpersist()
Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

wrapped_rdds()
Returns a list of the wrapped, Scala RDDs within each layer of the pyramid.

Returns [org.apache.spark.rdd.RDD]

2.13.11 geopyspark.geotrellis.neighborhood module

Classes that represent the various neighborhoods used in focal functions.

Note: Once a parameter has been entered for any one of these classes it gets converted to a `float` if it was originally an `int`.

class `geopyspark.geotrellis.neighborhood.`**`Circle`**(*radius*)

A circle neighborhood.

Parameters **`radius`** (*`int or float`*) – The radius of the circle that determines which cells fall within the bounding box.

`radius`

int or float – The radius of the circle that determines which cells fall within the bounding box.

`param_1`

float – Same as `radius`.

`param_2`

float – Unused param for `Circle`. Is 0.0.

`param_3`

float – Unused param for `Circle`. Is 0.0.

`name`

str – The name of the neighborhood which is, "circle".

Note: Cells that lie exactly on the radius of the circle are apart of the neighborhood.

class `geopyspark.geotrellis.neighborhood.`**`Wedge`**(*radius*, *start_angle*, *end_angle*)

A wedge neighborhood.

Parameters

- **`radius`** (*`int or float`*) – The radius of the wedge.
- **`start_angle`** (*`int or float`*) – The starting angle of the wedge in degrees.
- **`end_angle`** (*`int or float`*) – The ending angle of the wedge in degrees.

`radius`

int or float – The radius of the wedge.

`start_angle`

int or float – The starting angle of the wedge in degrees.

`end_angle`

int or float – The ending angle of the wedge in degrees.

`param_1`

float – Same as `radius`.

`param_2`

float – Same as `start_angle`.

`param_3`
float – Same as `end_angle`.

`name`
str – The name of the neighborhood which is, “wedge”.

class `geopyspark.geotrellis.neighborhood.`**`Nesw`**(*extent*)
A neighborhood that includes a column and row intersection for the focus.

Parameters **`extent`** (*`int or float`*) – The extent of this neighborhood. This represents the how many cells past the focus the bounding box goes.

`extent`
int or float – The extent of this neighborhood. This represents the how many cells past the focus the bounding box goes.

`param_1`
float – Same as `extent`.

`param_2`
float – Unused param for `Nesw`. Is 0.0.

`param_3`
float – Unused param for `Nesw`. Is 0.0.

`name`
str – The name of the neighborhood which is, “nesw”.

class `geopyspark.geotrellis.neighborhood.`**`Annulus`**(*inner_radius*, *outer_radius*)
An Annulus neighborhood.

Parameters

- **`inner_radius`** (*`int or float`*) – The radius of the inner circle.
- **`outer_radius`** (*`int or float`*) – The radius of the outer circle.

`inner_radius`
int or float – The radius of the inner circle.

`outer_radius`
int or float – The radius of the outer circle.

`param_1`
float – Same as `inner_radius`.

`param_2`
float – Same as `outer_radius`.

`param_3`
float – Unused param for `Annulus`. Is 0.0.

`name`
str – The name of the neighborhood which is, “annulus”.

2.13.12 geopyspark.geotrellis.ProtoBufCodecs module

`geopyspark.geotrellis.`**`protobuf`**
alias of *`geopyspark.geotrellis.protobuf`*

2.13.13 geopyspark.geotrellis.ProtoBufSerializer module

class geopyspark.geotrellis.protobufserializer.**ProtoBufSerializer**(*decoding_method*, *encoding_method*)

The serializer used by a RDD to encode/decode values to/from Python.

Parameters

- **decoding_method** (*func*) – The decocding function for the values within the RDD.
- **encoding_method** (*func*) – The encocding function for the values within the RDD.

decoding_method

func – The decocding function for the values within the RDD.

encoding_method

func – The encocding function for the values within the RDD.

dumps(*obj*)

Serialize an object into a byte array.

Note: When batching is used, this will be called with a list of objects.

Parameters **obj** – The object to serialized into a byte array.

Returns The byte array representation of the `obj`.

loads(*obj*)

Deserializes a byte array into a collection of Python objects.

Parameters **obj** – The byte array representation of an object to be deserialized into the object.

Returns A list of deserialized objects.

2.13.14 geopyspark.geotrellis.rasterize module

geopyspark.geotrellis.rasterize.**rasterize**(*geoms*, *crs*, *zoom*, *fill_value*, *cell_type=<CellType.FLOAT64: 'float64'>*, *options=None*, *partition_strategy=None*)

Rasterizes a Shapely geometries.

Parameters

- **geoms** (*[shapely.geometry] or (shapely.geometry) or pyspark.RDD[shapely.geometry]*) – Either a list, tuple, or a Python RDD of shapely geometries to rasterize.
- **crs** (*str or int*) – The CRS of the input geometry.
- **zoom** (*int*) – The zoom level of the output raster.
- **fill_value** (*int or float*) – Value to burn into pixels intersectiong geometry
- **cell_type** (str or *CellType*) – Which data type the cells should be when created. Defaults to `CellType.FLOAT64`.
- **options** (*RasterizerOptions*, optional) – Pixel intersection options.

- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy`, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will have the default `Partitioner` and a number of paritions that was determined by the method.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

`geopyspark.geotrellis.rasterize.`**`rasterize_features`**(*features*, *crs*, *zoom*, *cell_type=<CellType.FLOAT64: 'float64'>*, *options=None*, *zindex_cell_type=<CellType.INT8: 'int8'>*, *partition_strategy=None*)

Rasterizes a collection of *Feature*s.

Parameters

- **features** (*pyspark.RDD[*`Feature`*]*) – A Python `RDD` that contains *Feature*s.

  Note: The `properties` of each `Feature` must be an instance of *CellValue*.

- **crs** (*str or int*) – The CRS of the input geometry.
- **zoom** (*int*) – The zoom level of the output raster.

  Note: Not all rasterized `Features` may be present in the resulting layer if the `zoom` is not high enough.

- **cell_type** (str or *CellType*) – Which data type the cells should be when created. Defaults to `CellType.FLOAT64`.
- **options** (*RasterizerOptions*, optional) – Pixel intersection options.
- **zindex_cell_type** (str or *CellType*) – Which data type the `Z-Index` cells are. Defaults to `CellType.INT8`.
- **partition_strategy** (*HashPartitionStrategy* or `SpatialPartitioinStrategy`, optional) – Sets the `Partitioner` for the resulting layer and how many partitions it has. Default is, `None`.

  If `None`, then the output layer will have the default `Partitioner` and a number of paritions that was determined by the method.

  If `partition_strategy` is set but has no `num_partitions`, then the resulting layer will have the `Partioner` specified in the strategy with the with same number of partitions the source layer had.

  If `partition_strategy` is set and has a `num_partitions`, then the resulting layer will have the `Partioner` and number of partitions specified in the strategy.

Returns *TiledRasterLayer*

2.13.15 geopyspark.geotrellis.tms module

class `geopyspark.geotrellis.tms.`**`TileRender`**(*render_function*)

A Python implementation of the Scala geopyspark.geotrellis.tms.TileRender interface. Permits a callback from Scala to Python to allow for custom rendering functions.

Parameters **`render_function`** (*`Tile => PIL.Image.Image`*) – A function to convert geopyspark.geotrellis.Tile to a PIL Image.

`render_function`

Tile => PIL.Image.Image – A function to convert geopyspark.geotrellis.Tile to a PIL Image.

`renderEncoded`(*scala_array*)

A function to convert an array to an image.

Parameters **`scala_array`** – A linear array of bytes representing the protobuf-encoded contents of a tile

Returns bytes representing an image

class `geopyspark.geotrellis.tms.`**`TMS`**(*server*)

Provides a TMS server for raster data.

In order to display raster data on a variety of different map interfaces (e.g., leaflet maps, geojson.io, GeoNotebook, and others), we provide the TMS class.

Parameters **`server`** (*`JavaObject`*) – The Java TMSServer instance

`pysc`

pyspark.SparkContext – The `SparkContext` being used this session.

`server`

JavaObject – The Java TMSServer instance

`host`

str – The IP address of the host, if bound, else None

`port`

int – The port number of the TMS server, if bound, else None

`url_pattern`

string – The URI pattern for the current TMS service, with {z}, {x}, {y} tokens. Can be copied directly to services such as *geojson.io*.

`bind`(*host=None*, *requested_port=None*)

Starts up a TMS server.

Parameters

- **`host`** (*`str, optional`*) – The target host. Typically "localhost", "127.0.0.1", or "0.0.0.0". The latter will make the TMS service accessible from the world. If omitted, defaults to localhost.
- **`requested_port`** (*`optional, int`*) – A port number to bind the service to. If omitted, use a random available port.

classmethod `build`(*source*, *display*, *allow_overzooming=True*)

Builds a TMS server from one or more layers.

This function takes a SparkContext, a source or list of sources, and a display method and creates a TMS server to display the desired content. The display method is supplied as a ColorMap (only available when there is a single source), or a callable object which takes either a single tile input (when there is a single source) or a list of tiles (for multiple sources) and returns the bytes representing an image file for that tile.

Parameters

- **source** (tuple or orlist or *Pyramid*) – The tile sources to render. Tuple inputs are (str, str) pairs where the first component is the URI of a catalog and the second is the layer name. A list input may be any combination of tuples and `Pyramids`.
- **display** (ColorMap, *callable*) – Method for mapping tiles to images. ColorMap may only be applied to single input source. Callable will take a single numpy array for a single source, or a list of numpy arrays for multiple sources. In the case of multiple inputs, resampling may be required if the tile sources have different tile sizes. Returns bytes representing the resulting image.
- **allow_overzooming** (*bool*) – If set, viewing at zoom levels above the highest available zoom level will produce tiles that are resampled from the highest zoom level present in the data set.

host

Returns the IP string of the server's host if bound, else None.

Returns (str)

port

Returns the port number for the current TMS server if bound, else None.

Returns (int)

unbind()

Shuts down the TMS service, freeing the assigned port.

url_pattern

Returns the URI for the tiles served by the present server. Contains {z}, {x}, and {y} tokens to be substituted for the desired zoom and x/y tile position.

Returns (str)

2.13.16 geopyspark.geotrellis.union module

`geopyspark.geotrellis.union.`**union**(*layers*)

Unions togther two or more `RasterLayers` or `TiledRasterLayers`.

All layers must have the same `layer_type`. If the layers are `TiledRasterLayers`, then all of the layers must also have the same *TileLayout* and `CRS`.

Note: If the layers to be unioned share one or more keys, then the resulting layer will contain duplicates of that key. One copy for each instance of the key.

Parameters **layers** ([*RasterLayer*] or [*TiledRasterLayer*] or (*RasterLayer*) or (*TiledRasterLayer*)) – A colection of two or more `RasterLayers` or `TiledRasterLayers` layers to be unioned together.

Returns *RasterLayer* or *TiledRasterLayer*

2.14 geopyspark.vector_pipe package

class `geopyspark.vector_pipe.`**Feature**

Represents a geometry that is derived from an OSM Element with that Element's associated metadata.

Parameters

- **geometry** (*shapely.geometry*) – The geometry of the feature that is represented as a `shapely.geometry`. This geometry is derived from an OSM Element.
- **properties** (*Properties* or *CellValue*) – The metadata associated with the OSM Element. Can be represented as either an instance of `Properties` or a `CellValue`.

geometry
shapely.geometry – The geometry of the feature that is represented as a `shapely.geometry`. This geometry is derived from an OSM Element.

properties
Properties or *CellValue* – The metadata associated with the OSM Element. Can be represented as either an instance of `Properties` or a `CellValue`.

count(*value*) → integer – return number of occurrences of value

geometry
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

properties
Alias for field number 1

class geopyspark.vector_pipe.**Properties**
Represents the metadata of an OSM Element.

This object is one of two types that can be used to represent the `properties` of a *Feature*.

Parameters

- **element_id** (*int*) – The `id` of the OSM Element.
- **user** (*str*) – The display name of the last user who modified/created the OSM Element.
- **uid** (*int*) – The numeric id of the last user who modified the OSM Element.
- **changeset** (*int*) – The OSM `changeset` number in which the OSM Element was created/modified.
- **version** (*int*) – The edit version of the OSM Element.
- **minor_version** (*int*) – Represents minor changes between versions of an OSM Element.
- **timestamp** (*datetime.datetime*) – The time of the last modification to the OSM Element.
- **visible** (*bool*) – Represents whether or not the OSM Element is deleted or not in the database.
- **tags** (*dict*) – A `dict` of `str`s that represents the given features of the OSM Element.

element_id
int – The `id` of the OSM Element.

user
str – The display name of the last user who modified/created the OSM Element.

uid
int – The numeric id of the last user who modified the OSM Element.

changeset
int – The OSM `changeset` number in which the OSM Element was created/modified.

version
int – The edit version of the OSM Element.

minor_version
int – Represents minor changes between versions of an OSM Element.

timestamp
datetime.datetime – The time of the last modification to the OSM Element.

visible
bool – Represents whether or not the OSM Element is deleted or not in the database.

tags
dict – A `dict` of `str`s that represents the given features of the OSM Element.

changeset
Alias for field number 3

count(*value*) → integer – return number of occurrences of value

element_id
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

minor_version
Alias for field number 5

tags
Alias for field number 8

timestamp
Alias for field number 6

uid
Alias for field number 2

user
Alias for field number 1

version
Alias for field number 4

visible
Alias for field number 7

class geopyspark.vector_pipe.**CellValue**
Represents the `value` and `zindex` of a geometry.

This object is one of two types that can be used to represent the `properties` of a *`Feature`*.

Parameters

- **value** (`int or float`) – The value of all cells that intersects the associated geometry.
- **zindex** (`int`) – The `Z-Index` of each cell that intersects the associated geometry. `Z-Index` determines which value a cell should be if multiple geometries intersect it. A high `Z-Index` will always be in front of a `Z-Index` of a lower value.

value
int or float – The value of all cells that intersects the associated geometry.

zindex
int – The `Z-Index` of each cell that intersects the associated geometry. `Z-Index` determines which value a cell should be if multiple geometries intersect it. A high `Z-Index` will always be in front of a `Z-Index` of a lower value.

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

value
Alias for field number 0

zindex
Alias for field number 1

2.14.1 geopyspark.vector_pipe.features_collection module

class `geopyspark.vector_pipe.features_collection.`**FeaturesCollection**(*scala_features*)
Represents a collection of features from OSM data. A `feature` is a geometry that is derived from an OSM Element with that Element's associated metadata. These `features` are grouped together by their geometry type.

There are 4 different types of geometries that a `feature` can contain:

- `Points`
- `Lines`
- `Polygons`
- `MultiPolygons`

Parameters **scala_features** (*py4j.JavaObject*) – The Scala representation of `FeaturesCollection`.

scala_features
py4j.JavaObject – The Scala representation of `FeaturesCollection`.

get_line_features_rdd()
Returns each `Line feature` in the `FeaturesCollection` as a *`Feature`* in a Python RDD.

Returns `RDD[Feature]`

get_line_tags()
Returns all of the unique tags for all of the `Lines` in the `FeaturesCollection` as a `dict`. Both the keys and values of the `dict` will be `strs`.

Returns dict

get_multipolygon_features_rdd()
Returns each `MultiPolygon feature` in the `FeaturesCollection` as a *`Feature`* in a Python RDD.

Returns `RDD[Feature]`

get_multipolygon_tags()
Returns all of the unique tags for all of the `MultiPolygons` in the `FeaturesCollection` as a `dict`. Both the keys and values of the `dict` will be `strs`.

Returns dict

get_point_features_rdd()

Returns each `Point feature` in the `FeaturesCollection` as a *`Feature`* in a Python RDD.

Returns `RDD[Feature]`

get_point_tags()

Returns all of the unique tags for all of the `Points` in the `FeaturesCollection` as a `dict`. Both the keys and values of the `dict` will be `strs`.

Returns dict

get_polygon_features_rdd()

Returns each `Polygon feature` in the `FeaturesCollection` as a *`Feature`* in a Python RDD.

Returns `RDD[Feature]`

get_polygon_tags()

Returns all of the unique tags for all of the `Polygons` in the `FeaturesCollection` as a `dict`. Both the keys and values of the `dict` will be `strs`.

Returns dict

2.14.2 geopyspark.vector_pipe.osm_reader module

`geopyspark.vector_pipe.osm_reader.`**from_orc**(*source*)

Reads in OSM data from an orc file that is located either locally or on S3. The resulting data will be read in as an instance of *`FeaturesCollection`*.

Parameters **source** (*str*) – The path or URI to the orc file to be read. Can either be a local file, or a file on S3.

Note: Reading a file from S3 requires additional setup depending on the environment and how the file is being read.

The following describes the parameters that need to be set depending on how the files are to be read in. However, **if reading a file on EMR, then the access key and secret key do not need to be set**.

If using `s3a://`, then the following `SparkConf` parameters need to be set:

- `spark.hadoop.fs.s3a.impl`
- `spark.hadoop.fs.s3a.access.key`
- `spark.hadoop.fs.s3a.secret.key`

If using `s3n://`, then the following `SparkConf` parameters need to be set:

- `spark.hadoop.fs.s3n.access.key`
- `spark.hadoop.fs.s3n.secret.key`

An alternative to passing in your S3 credentials to `SparkConf` would be to export them as environment variables:

- `AWS_ACCESS_KEY_ID=YOUR_KEY`
- `AWS_SECRET_ACCESS_KEY_ID=YOUR_SECRET_KEY`

Returns *`FeaturesCollection`*

`geopyspark.vector_pipe.osm_reader.`**`from_dataframe`**`(`*dataframe*`)`

Reads OSM data from a Spark `DataFrame`. The resulting data will be read in as an instance of *`FeaturesCollection`*.

Parameters **`dataframe`** (*`DataFrame`*) – A Spark `DataFrame` that contains the OSM data.

Returns *`FeaturesCollection`*

Python Module Index

Index