GeoPySpark Documentation

Release 0.1.0

Jacob Bouffard, James McClean, Eugene Cheipesh

Jan 30, 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 challanges. 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

A Quick Example

Here is a quick example of GeoPySpark. In the following code, we take NLCD data of the state of Pennslyvania from 2011, and do a polygonal summary of an area of interest to find the min and max classifcations values of that area.

If you wish to follow along with this example, you will need to download the NLCD data and the geojson that represents the area of interest. Running these two commands will download these files for you:

```
curl -o /tmp/NLCD2011_LC_Pennsylvannia.zip https://s3-us-west-2.amazonaws.com/prd-tnm/
↪StagedProducts/NLCD/2011/landcover/states/NLCD2011_LC_Pennsylvania.zip?ORIG=513_
↪SBDDG
unzip /tmp/NLCD2011_LC_Pennsylvannia.zip
curl -o /tmp/area_of_interest.geojson https://s3.amazonaws.com/geopyspark-test/area_
↪of_interest.json
```

```
import json
from functools import partial

from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis.constants import SPATIAL, ZOOM
from geopyspark.geotrellis.geotiff_rdd import get
from geopyspark.geotrellis.catalog import write

from shapely.geometry import Polygon, shape
from shapely.ops import transform
import pyproj


# Create the GeoPyContext
geopysc = GeoPyContext(appName="example", master="local[*]")

# Read in the NLCD tif that has been saved locally.
# This tif represents the state of Pennsylvania.
raster_rdd = get(geopysc=geopysc, rdd_type=SPATIAL,
uri='/tmp/NLCD2011_LC_Pennsylvania.tif',
options={'numPartitions': 100})
```

```
tiled_rdd = raster_rdd.to_tiled_layer()

# Reproject the reclassified TiledRasterRDD so that it is in WebMercator
reprojected_rdd = tiled_rdd.reproject(3857, scheme=ZOOM).cache().repartition(150)

# We will do a polygonal summary of the north-west region of Philadelphia.
with open('/tmp/area_of_interest.json') as f:
    txt = json.load(f)

geom = shape(txt['features'][0]['geometry'])

# We need to reporject the geometry to WebMercator so that it will intersect with
# the TiledRasterRDD.
project = partial(
    pyproj.transform,
    pyproj.Proj(init='epsg:4326'),
    pyproj.Proj(init='epsg:3857'))

area_of_interest = transform(project, geom)

# Find the min and max of the values within the area of interest polygon.
min_val = reprojected_rdd.polygonal_min(geometry=area_of_interest, data_type=int)
max_val = reprojected_rdd.polygonal_max(geometry=area_of_interest, data_type=int)

print('The min value of the area of interest is:', min_val)
print('The max value of the area of interest is:', max_val)

# We will now pyramid the relcassified TiledRasterRDD so that we can use it in a TMS␣
↪server later.
pyramided_rdd = reprojected_rdd.pyramid(start_zoom=1, end_zoom=12)

# Save each layer of the pyramid locally so that it can be accessed at a later time.
for pyramid in pyramided_rdd:
    write('file:///tmp/nld-2011', 'pa', pyramid)
```

CHAPTER 3

Contact and Support

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

3.1 Changelog

3.1.1 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`.
- Eublidean distance added to `TiledRasterRDD`.
- Neighborhoods submodule added to make focal operations easier.

geopyspark.command

- GeoPySpark can now be used 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 data.
- Ingesting and creating a tile server with data from Sentinel.

3.2 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.

3.2.1 Building GeoPySpark

1. Install and setup Hadoop (the master branch is currently built with 2.0.1).
2. Check out this repository.
3. Pick the branch corresponding to the version you are targeting
4. Run `make install` to build GeoPyspark.

3.2.2 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.

3.2.3 Git Branching Model

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

3.2.4 Git Commit Messages

We follow the 'imperative present tense' style for commit messages. (e.g. "Add new EnterpriseWidgetLoader instance")

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.2.8 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.

3.3 Core Concepts

3.3.1 Dealing with GeoTrellis Types

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.

You may notice as read through this section that camel case is used instead of Python's more traditional naming convention for some values. This is because Scala uses this style of naming, and when it receives data from Python it expects the value names to be in camel case.

Raster

GeoPySpark differs in how it represents rasters from other geo-spatial Python libraries like rasterio. In GeoPySpark, they are represented as a `dict`.

The fields used to represent rasters:

- **no_data_value**: The value that represents no data in raster. This can be represented by a variety of types depending on the value type of the raster.
- **data** (nd.array): The raster data itself. It is contained within a NumPy array.

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

ProjectedExtent

Describes both the area on Earth a raster represents in addition to its CRS. In GeoPySpark, this is represented as a `dict`.

The fields used to represent `ProjectedExtent`:

- **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.

Example:

```
extent = Extent(0.0, 1.0, 2.0, 3.0)

# using epsg
epsg_code = 3857
projected_extent = {'extent': extent, 'epsg': epsg}

# using proj4
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 "
projected_extent = {'extent': extent, 'proj4': proj4}
```

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

TemporalProjectedExtent

Describes the area on Earth the raster represents, its CRS, and the time the data was collected. In GeoPySpark, this is represented as a `dict`.

The fields used to represent `TemporalProjectedExtent`.

- **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.
- **instance** (int): The time stamp of the raster.

Example:

```
extent = Extent(0.0, 1.0, 2.0, 3.0)

epsg_code = 3857
instance = 1.0
projected_extent = {'extent': extent, 'epsg': epsg, 'instance': instance}
```

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

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. In GeoPySpark, this is represented as a `dict`.

The fields used to represent `SpatialKey`:

- **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.

Example:

```
spatial_key = {'col': 0, 'row': 0}
```

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. In GeoPySpark, this is represented as a `dict`.

The fields used to reprsent `SpaceTimeKey`:

- **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.
- **instance** (int): The time stamp of the raster.

Example:

```
spatial_key = {'col': 0, 'row': 0, 'instant': 0.0}
```

3.3.2 How Data is Stored in RDDs

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

GeoPySpark does not work with PySpark RDDs, but rather, uses Python classes that are wrappers of classes in Scala that contain and work with a Scala RDD. The exact workings of this relationship between the Python and Scala classes will not be discussed in this guide, instead the focus will be on what these Python classes represent and how they are used within GeoPySpark.

All RDDs in GeoPySpark contain tuples, which will be referred to in this guide as `(K, V)`. `V` will always be a raster, but `K` differs depending on both the wrapper class and the nature of the data itself.

Where is the Actual RDD?

The actual RDD that is being worked on exists in Scala. Even if the RDD was originally created in Python, it will be serialized and sent over to Scala where it will be decoded into a Scala RDD.

None of the operations performed on the RDD occur in Python, and the only time the RDD will be moved to Python is if the user decides to bring it over.

RasterRDD

`RasterRDD` is one of the two wrapper classes in GeoPySpark and deals with untiled data. What does it mean for data to be untiled? It means that each element within the RDD has not been modified in such a way that would make it a part of a larger, overall layout. For example, a distributed collection of rasters of a contiguous area could be derived from GeoTiffs of different sizes. This, in turn, could mean that there's a lack of uniformity when viewing the area as a whole. It is this, "raw" data that is stored within `RasterRDD`.

It would help to have all of the data uniform when working with it, and that is what `RasterRDD` accomplishes. The point of this class is to format the data within the RDD to a specified layout.

As mentioned in the previous section, both wrapper classes hold data in tuples. With the `K` of each tuple being different between the two. In the case of `RasterRDD`, `K` is either `ProjectedExtent` or `TemporalProjectedExtent`.

TiledRasterRDD

`TiledRasterRDD` is the second of the two wrapper classes in GeoPySpark and deals with tiled data. Which means the rasters inside of the RDD have been fitted to a certain layout. 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 perform map algebra, pyramid, and save the RDD among other operations.

As mentioned in the previous section, both wrapper classes hold data in tuples. With the `K` of each tuple being different between the two. In the case of `TiledRasterRDD`, `K` is either `SpatialKey` or `SpaceTimeKey`.

3.4 GeoPyContext

`GeoPyContext` is a class that acts as a wrapper for `SparkContext` in GeoPySpark. Why have such a class? It has to do with how the Python and Scala code communicate with one another. By hooking into the JVM, the Python side is able to access classes, objects, and functions from Scala. This also holds true for Scala, which is able to send over values to Python.

Before being sent to the other side, though, the values must be formatted in such a way so they can be serialized/deserialzed. `GeoPyContext` makes this a little easier by providing methods that will prepare the data before it is sent over. This is why a `GeoPyContext` instance is needed for almost all functions and class constructors.

3.4.1 Initializing GeoPyContext

Initializing `GeoPyContext` can be done through two different methods: either by giving it an exisiting `SparkContext`, or by passing in the arguments used to construct a `SparkContext`.

```
from geopyspark.geopycontext import GeoPyContext

from pyspark import SparkContext

# Using an existing SparkContext.
sc = SparkContext(appName="example", master="local[*]")

geopysc = GeoPyContext(sc)

# Using SparkContext args.
geopysc = GeoPyContext(appName="example", master="local[*]")
```

3.5 RasterRDD and TiledRasterRDD

This section seeks to explain how to create and use `RasterRDD` and `TiledRasterRDD`. Before continuing this example, it is suggested that you read *How Data is Stored in RDDs*.

3.5.1 Creating RasterRDD and TiledRasterRDD

RasterRDD

Of the two different RDD classes, `RasterRDD` has the least number of ways to be initialized. There are just two: through reading GeoTiffs from the local file system, S3, or HDFS; or from an existing PySpark RDD.

From GeoTiffs

The *get()* method in `geopyspark.geotrellis.geotiff_rdd` creates an instance of `RasterRDD` from GeoTiffs.

```
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis.constants import SPATIAL
from geopyspark.geotrellis.geotiff_rdd import get

geopysc = GeoPyContext(appName="rasterrdd-example", master="local")

raster_rdd = get(geopysc=geopysc, rdd_type=SPATIAL, "path/to/your/geotiff.tif")
```

Note: If you have multiple GeoTiffs, you can just specify the directory where they're all stored. Or if the GeoTiffs are spread out in multiple locations, you can give `get` a `list` of the places to read in the GeoTiffs.

From PySpark RDDs

The second option is to create a new `RasterRDD` from a PySpark RDD via the *from_numpy_rdd()* class method. This step is a bit more involved than the last, as it requires the data within the PySpark RDD to be formatted in a specific way.

```
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis import Extent
from geopyspark.geotrellis.constants import SPATIAL
from geopyspark.geotrellis.rdd import RasterRDD

import numpy as np

geopysc = GeoPyContext(appName="rasterrdd-example", master="local")

arr = np.ones((1, 16, 16), dtype=int)

# The raster data that will be contained in this RasterRDD will be 16x16,
# and will have a noData value of -500.
tile = {'no_data_value': -500, 'data': arr}

extent = Extent(0.0, 1.0, 2.0, 3.0)

# Since the RasterRDD will be SPATIAL, a ProjectedExtent is constructed.
projected_extent = {'extent': extent, 'epsg': 3857}

# Create a PySpark RDD that contains a single tuple, (projected_extent, tile)
# Note: The order of the values in the tuple is important. ProjectedExtent
# or TemporalProjectedExtent MUST Be the first element.
rdd = geopysc.pysc.parallelize([(projected_extent, tile)])

raster_rdd = RasterRDD.from_numpy_rdd(geopysc=geopysc, rdd_type=SPATIAL, numpy_
↪rdd=rdd)
```

TiledRasterRDD

Unlike `RasterRDD`, `TiledRasterRDD` has multiple ways of being created.

From PySpark RDDs

`TiledRasterRDD` also has the class method, *`from_numpy_rdd()`*.

```
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis import Extent, TileLayout, Bounds, LayoutDefinition
from geopyspark.geotrellis.constants import SPATIAL
from geopyspark.geotrellis.rdd import TiledRasterRDD

import numpy as np

geopysc = GeoPyContext(appName="tiledrasterrdd-example", master="local")

data = 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]]])

# Data to be placed within the TiledRasterRDD.
# Each value is a tuple where the first value is either a SpatialKey or a
# SpaceTime. With the second being the tile.
layer = [({'row': 0, 'col': 0}, {'no_data_value': -1.0, 'data': data}),
         ({'row': 1, 'col': 0}, {'no_data_value': -1.0, 'data': data}),
         ({'row': 0, 'col': 1}, {'no_data_value': -1.0, 'data': data}),
         ({'row': 1, 'col': 1}, {'no_data_value': -1.0, 'data': data})]

# Creating the PySpark RDD.
rdd = BaseTestClass.geopysc.pysc.parallelize(layer)

# All TiledRasterRDDs have metadata that describes the layout of data within
# it. Therefore, in order to create it from a PySpark RDD, the metadata must
# be either created, or taken from elsewhere.
extent = Extent(0.0, 0.0, 33.0, 33.0)
layout = TileLayout(2, 2, 5, 5)
bounds = Bounds({'col': 0, 'row': 0}, {'col': 1, 'row': 1})
layout_definition = LayoutDefinition(extent, layout)

metadata = Metadata(
    bounds=bounds,
    crs='+proj=longlat +datum=WGS84 +no_defs ',
    cell_type='float32ud-1.0',
    extent=extent,
    layout_definition=layout_definition)

tiled_rdd = TiledRasterRDD.from_numpy_rdd(geopysc=geopysc, rdd_type=SPATIAL,
                                          numpy_rdd=rdd, metadata=metadata)
```

Through Rasterization

Another means of producing `TiledRasterRDD` is through rasterizing a Shapely geometry via the *`rasterize()`* method.

```
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis import Extent
from geopyspark.geotrellis.constants import SPATIAL
from geopyspark.geotrellis.rdd import TiledRasterRDD

from shapely.geometry import Polygon

geopysc = GeoPyContext(appName="tiledrasterrdd-example", master="local")

extent = Extent(0.0, 0.0, 11.0, 11.0)

polygon = Polygon([(0, 11), (11, 11), (11, 0), (0, 0)])

# Creates a TiledRasterRDD from a Shapely Polygon. The resulting raster will
# be 256x256 and all values within it are 1.
tiled_rdd = TiledRasterRDD.rasterize(geopysc=geopysc, rdd_type=SPATIAL,
                                     geometry=polygon, extent=extent,
                                     cols=256, rows=256, fill_value=1)
```

Through Euclidean Distance

The final way to create `TiledRasterRDD` is by calculating the Euclidean of a Shapely geometry. `euclidean_distance()` is the class method which does this. While you can use any geometry to perform Euclidean distance, it is recommended **not** to use Polygons if they cover many cells of the resulting raster. As this can impact performance in a negative way.

```
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis import Extent
from geopyspark.geotrellis.constants import SPATIAL
from geopyspark.geotrellis.rdd import TiledRasterRDD

from shapely.geometry import MultiPoint
import pyproj

geopysc = GeoPyContext(appName="tiledrasterrdd-example", master="local")

# Shapely produces points in LatLng by default. However, GeoPySpark tends to
# work with values in WebMercator, so we must reproject the geometries.
latlong = pyproj.Proj(init='epsg:4326')
webmerc = pyproj.Proj(init='epsg:3857')
points = MultiPoint([pyproj.transform(latlong, webmerc, 1, 1),
                     pyproj.transform(latlong, webmerc, 2, 2)])

# Makes a TiledRasterRDD from the Euclidean distance calculation.
# The resulting TiledRasterRDD will have a zoom level of 7.
tiled_rdd = TiledRasterRDD.euclidean_distance(geopysc=geopysc,
                                              geometry=points,
                                              source_crs=3857,
                                              zoom=7)
```

3.5.2 Using RasterRDD and TiledRasterRDD

After initializing `RasterRDD` and/or `TiledRasterRDD`, it is now time to use them.

Common Methods

While different in terms of functionality, `RasterRDD` and `TiledRasterRDD` both share some methods.

Converting to a PySpark RDD

If you wish to you convert to a PySpark RDD, it can be done via the `to_numpy_rdd` method.

```
# RasterRDD
raster_rdd.to_numpy_rdd()

# TiledRasterRDD
tiled_rdd.to_numpy_rdd()
```

Reclassifying Values

`reclassify` can reclassify values in either `RasterRDD` or `TiledRasterRDD`. This is done by binning each value in the RDD.

The `boundary_startegy` will determine how each value will be binned. These are the strategies to choose from: `GREATERTHAN`, `GREATERTHANOREQUALTO`, `LESSTHAN`, `LESSTHANOREQUALTO`, and `EXACT`.

If a value does not fall within the boundary, then it's given the `no_data_value`. A different replacement can be used instead with `replace_nodata_with`.

```
from geopyspark.geotrellis.constants import EXACT, LESSTHAN

value_map = {1: 0}
# All values less than or equal to 1 will now become zero.
# Any other number is now whatever the no_data_value is for this
# TiledRasterRDD
tiled_rdd.reclassify(value_map=value_map, data_type=int)

value_map = {5.0: 10.0, 15.0: 20.0}

# Only 5.0 and 15.0 will be reclassified. Everything else will become -1000.0
tiled_rdd.relcassify(value_map=value_map, data_type=float, boundary_strategy=EXACT,
                     replace_no_data_with=-1000.0)
```

Min and Max

`get_min_max` will produce the min and max values of the RDD. They always be returned as `floats`. Regardless of the type of the values.

```
tiled_rdd.get_min_max()
```

RasterRDD

The purpose of RasterRDD is store and format data to produce a TiledRasterRDD. Thus, this class lacks the methods needed to perform any kind of spatial analysis. It can be thought of as something of an "organizer". Which sorts and lays out the data so that TiledRasterRDD can perform operations on the data.

Collecting Metadata

In order to convert a RasterRDD to a TiledRasterRDD the *Metadata* must first be collected; as it contains the information on how the data should be formatted and laid out in the TiledRasterRDD. *collect_metadata()* is used to obtain the metadata, and it can accept to different types of inputs depending on how one wishes to layout the data.

The first option is to specify an *Extent* and a *TileLayout* for the Metadata. Where the Extent is the area that will be covered by the Tiles and the TileLayout describes the Tiles and the grid they're arranged on.

```
from geopyspark.geotrellis import Extent, TileLayout

extent = Extent(0.0, 0.0, 33.0, 33.0)
tile_layout = TileLayout(2, 2, 256, 256)

# The Metadata that will be returned will conform to the extent and tile
# layout that was given. In this case, the rasters will be tiled into a 2x2
# grid with each Tile having 256 cols and rows. This grid will cover the
# area within the extent.
md = raster_rdd.collect_metadata(extent=extent, layout=tile_layout)
```

The other option is to simply give collect_metadata the tile_size that each Tile should be in the resulting grid. Extent and TileLayout will be calculated from this size. Using this method will ensure that the native resolutions of the rasters are kept.

```
# tile_size has a default value of 256. If this works for your case, then
# you can just do this
md = raster_rdd.collect_metadata()

# Otherwise, you can specify your own tile_size.
md = raster_rdd.collect_metadata(tile_size=512)
```

Formatting the Data to a Layout

Once Metadata has been obtained, RasterRDD will be able to format the data, which will result in a new TiledRasterRDD instance. There are two methods to do this: *cut_tiles()* and *tile_to_layout()*.

Both of these methods have the same inputs and similar outputs, however, there is one key difference between the two. cut_tiles will cut the rasters to the given layout, but will not fix any overlap that may occur. Whereas tile_to_layout will cut and then merge together areas that are overlapped. This matters as each Tile is referenced by a key, and if there's overlap than there could be duplicate keys.

Therefore, it is recommended to use tile_to_layout to ensure there is no duplication.

```
md = raster_rdd.collect_metadata()
tiled_rdd = raster_rdd.tile_to_layout(layer_metadata=md)

# resample_method can be set when doing the formatting. For this example,
# BILINEAR will be used. The defatul method is NEARESTNEIGHBOR.
```

```
from geopyspark.geotrellis.constants import BILINEAR

tiled_rdd = raster_rdd.tile_to_layout(layer_metadata=md, resample_method=BILINEAR)
```

A Quicker Way to TiledRasterRDD

to_tiled_layer() allows the user to layout their data and produce a `TiledRasterRDD` in just one step. This method is `collect_metadata` and `tile_to_layout` combined, and is used to save a little time when writing.

```
# Using Extent and TileLayout

from geopyspark.geotrellis import Extent, TileLayout

extent = Extent(0.0, 0.0, 33.0, 33.0)
tile_layout = TileLayout(2, 2, 256, 256)

tiled_rdd = raster_rdd.to_tiled_layer(extent=extent, layout=tile_layout)

# Or using tile_size instead

tiled_rdd = raster_rdd.to_tiled_layer()
```

TiledRasterRDD

`TiledRasterRDD` will be the class that will see the most use. It provides all the methods needed to perform a computations and analysis on the data. When reading and saving layers, this class will be used.

A Note on Using Geometries

Before doing operations that involve geometries, it is important to check to make sure that the geometry is in the correct projection. Geometries created through Shapely are in LatLong. Unless the data in `TiledRasterRDD` is also in this projection, the geometry being used will need to be reprojected.

```
from functools import partial

from shapely.geometry import Polygon
from shapely.ops import transform
import pyproj

polygon = Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)])

# Reporjects the geometry to WebMercator so that it will intersect with
# the TiledRasterRDD.
project = partial(
    pyproj.transform,
    pyproj.Proj(init='epsg:4326'),
    pyproj.Proj(init='epsg:3857'))

reprojected_polygon = transform(project, geom)
```

Reprojecting

Often the tiles within `TiledRasterRDD` will have to be reprojected. There is a method to do this aptly named, *`reproject()`*. If you wish to create a TMS server from this data, then this method should be used to ensure that the layout will work when pyramiding (more on that in a bit).

If you do not wish to create a TMS server, and just want to reproject the data, then there are two different ways to different ways to do so.

```
# Using Extant and TileLayout

from geopyspark.geotrellis import Extent, TileLayout

extent = Extent(0.0, 0.0, 33.0, 33.0)
tile_layout = TileLayout(2, 2, 256, 256)

reprojected_rdd = tiled_rdd.reproject(target_crs=3857, extent=extent,
                                      layout=tile_layout)

# Using tile_size

reprojected_rdd = tiled_rdd.reproject(target_crs=3857)
```

If you want to make a TMS server, then there is only one option available for reprojecting.

```
from geopyspark.geotrellis.constants import ZOOM

reprojected_rdd = tiled_rdd.reproject(target_crs=3857, scheme=ZOOM)

# Reprojecting with different tile_size

reprojected_rdd = tiled_rdd.reproject(target_crs=3857, scheme=ZOOM, tile_size=512)
```

What is the difference between using and not using `ZOOM`? It has to do with how GeoTrellis represents the layout of the data in the RDD. There are three different classes GeoTrellis uses: `LayoutDefinition`, `FloatingLayoutScheme` and `ZoomedLayoutScheme`. The exact nature and differences between these classes will not be discussed here, rather, a brief explanation will be given.

Because the resolution of images changes as one zooms in and out when using a TMS server, the layout of the tiles changes. Neither `LayoutDefinition` or `FloatingLayoutScheme` have the ability to adjust the layout from a zoom. Only `ZoomedLayoutScheme` can do this, which is why it must be set when reprojecting.

Retiling

It is possible to change the layout of the tiles within `TiledRasterRDD` via *`tile_to_layout()`*.

```
from geopyspark.geotrellis import Extent, TileLayout, LayoutDefinition

extent = Extent(100.0, 100.0, 250.0, 250.0)
tile_layout = TileLayout(5, 5, 256, 256)
layout_definition = TileDefinition(extent, tile_layout)

retiled_rdd = tiled_rdd.tile_to_layout(layout=layout_definition)
```

Masking

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

```
 from shapely.geometry import Polygon

polygon = Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)])

# The resulting TiledRasterRDD will only contain values that were interested
# by this Polygon

masked_rdd = tiled_rdd.mask(geometries=polygon)
```

Stitching

Using *stitch()* will produce a single raster by stitching together all of the tiles within the TiledRasterRDD. This can only be done with SPATIAL RDDs, and is not recommended if the data contained within is large. As it can cause crashes due to its size.

```
raster = tiled_rdd.stitch()
```

Pyramiding

Before creating a TMS server, a TiledRasterRDD needs to be pyramided first. *pyramid()* will create a new TiledRasterRDD for each zoom level, and the resulting list can then be either be accessed to fetch specific tiles or can be saved for later use.

```
# Creates 12 new TiledRasterRDDs where each one has a different layout
# depending on its zoom level.
pyramided_rdds = tiled_rdd.pyramid(start_zoom=12, end_zoom=1)
```

Why is start_zoom greater than end_zoom? This is because start_zoom represents the lowest or most zoomed level of the pyramid. And the pyramiding process starts with the greatest zoom and works its way up to the most zoomed out.

Operations

TiledRasterRDDs can perform both local and focal operations.

Local

Performing local operations with TiledRasterRDDs can be performed with ints, floats, or other TiledRasterRDDs.

```
# All values will have one added to them
tiled_rdd + 1

# Find the average of two TiledRasterRDDs
(tiled_rdd_1 + tiled_rdd_2) / 2

# The position of TiledRasterRDD in the operation doesn't matter, so it can
```

```
# be used on either side of of the operation.
1 / (5 - tiled_rdd)
```

Focal

Focal operations are done by selecting both a `neighborhood` and a `operation`. Because the inputs must be sent over to Scala, the `operation` must be entered in the form of a constant.

The values used to represent `operation` are: `SUM`, `MIN`, `MAX`, `MEAN`, `MEDIAN`, `MODE`, `STANDARDDEVIATION`, `ASPECT`, and `SLOPE`. These are all of the current available focal operations that can be done in GeoPySpark.

`neighborhood` can be specified with either a `Neighboorhod` sub-class, or a constant.

```
from geopyspark.geotrellis.neighborhoods import Square
from geopyspakr.geotrellis.constants import SLOPE

# Creates a Square neighborhood. Setting extent to 1 will mean that only one
# cell past the focus of the bounding box will be included in the
# neighborhood. Thus it creates a neighborhood that is 3x3 cells in size.
square_neighborhood = Square(extent=1)

# Calculate the slope for each neighborhood in the TiledRasterRDD
slope_rdd = tiled_rdd.focal(operation=SLOPE, neighborhood=square_neighborhood)


# To perform a focal operation with creating a Neighborhood class.

from geopyspark.geotrellis.constants import SQUARE

# Since a class wasn't initialized, the parameters to make the neighborhood
# must be passed in to the method. Square only requires one parameter, so
# only param_1 needs to be set.
slope_rdd = tiled_rdd.focal(operation=SLOPE, neighborhood=SQUARE, param_1=1)
```

Polygonal Summary Methods

In addition to local and focal methods, `TiledRasterRDD` can also perform polygonal summary methods. Using Shapely geometries, one can find the min, max, sum, and mean of all of the values intersected by the geometry.

```
from shapely.geometry import Polygon

polygon = Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)])

# Finds the min value that falls inside the Polygon. The data type of the
# values within the Tiles must be stated. For this example, they are ints.
tiled_rdd.polygonal_min(geometry=polygon, data_type=int)

# Finds the max value that falls inside the Polygon.
tiled_rdd.polygonal_max(geometry=polygon, data_type=float)

# Finds the sum of the values that fall inside the Polygon.
tiled_rdd.polygonal_sum(geometry=polygon, data_type=int)

# polygonal_mean will always return a float, so there's no need to set
```

```
# data_type.
tiled_rdd.polygonal_mean(geometry=polygon)
```

Cost Distance

It's possible to calculate the cost distance of a `TiledRasterRDD` via *`cost_distance()`*.

```
from shapely.geometry import Point

points = [Point(0, 0), Point(1, 2)]

tiled_rdd.cost_distance(geometries=points, max_distance=144000)
```

3.6 Catalog

The `Catalog` class allows the user to read, write, and query GeoTrellis layers in GeoPySpark. Because GeoPySpark is a Python binding of GeoTrellis, the layers it saves are GeoTrellis layers.

3.6.1 Accessing the Data

GeoPySpark supports various backends to read and save data to and from. These are the current backends supported:

- **Local Filesystem**
- **HDFS**
- **S3**
- **Cassandra**
- **HBase**
- **Accumulo**

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

- **Local Filesystem**: file://my_folder/my_catalog/
- **HDFS**: hdfs://my_folder/my_catalog/
- **S3**: s3://my_bucket/my_catalog/
- **Cassandra**: cassandra:name?username=user&password=pass&host=host1&keyspace=key&table=table
- **HBase**: hbase://zoo1, zoo2: port/table
- **Accumulo**: accumulo://username:password/zoo1, zoo2/instance/table

It is important to note that neither HBase or Accumulo have native support for `URI`s. Thus, GeoPySpark uses its own pattern for these two systems.

The `URI` for HBase follows this pattern:

- hbase://zoo1, zoo2, . . . , zooN: port/table

The `URI` for Accumulo follows this pattern:

- accumulo://username:password/zoo1, zoo2/instance/table

Some backends require various options to be set, and each function in `Catalog` has an `options` parameter where they can specified. These are the backends that need additional values and the options to set for each.

Fields that can be set for Cassandra:

- **replicationStrategy** (str, optional): If not specified, then 'SimpleStrategy' will be used.
- **replicationFactor** (int, optional): If not specified, then 1 will be used.
- **localDc** (str, optional): If not specified, then 'datacenter1' will be used.
- **usedHostsPerRemoteDc** (int, optional): If not specified, then 0 will be used.
- **allowRemoteDCsForLocalConsistencyLevel (int, optional): If you'd like this feature,** then the value would be 1, Otherwise, the value should be 0. If not specified, then 0 will be used.

Fields that can be set for HBase:

- **master** (str, optional): If not specified, then `null` will be used.

A Note on the Formatting of Rasters

A small, but important, note needs to be made about how rasters that are saved and/or read in are formatted in GeoPySpark. All rasters 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.

3.7 Ingesting a Grayscale Image

This example shows how to ingest a grayscale image and save the resultsr locally.

3.7.1 The Code

Here's the code to run the ingest.

```
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis.constants import SPATIAL, ZOOM
from geopyspark.geotrellis.catalog import write
from geopyspark.geotrellis.geotiff_rdd import get

geopysc = GeoPyContext(appName="python-ingest", master="local[*]")

# Read the GeoTiff from S3
rdd = get(geopysc, SPATIAL, "file:///tmp/cropped.tif")

metadata = rdd.collect_metadata()

# tile the rdd to the layout defined in the metadata
laid_out = rdd.tile_to_layout(metadata)

# reproject the tiled rasters using a ZoomedLayoutScheme
reprojected = laid_out.reproject("EPSG:3857", scheme=ZOOM)

# pyramid the TiledRasterRDD to create 12 new TiledRasterRDDs
# one for each zoom level
pyramided = reprojected.pyramid(start_zoom=12, end_zoom=1)

# Save each TiledRasterRDDs locally
```

```
for tiled in pyramided:
    write("file:///tmp/python-catalog", "python-ingest", tiled)
```

Running the Code

Before you can run this example, the example file will have to be downloaded. Run this command to save the file locally in the `/tmp` directory.

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

Running the code is simple, and you have two different ways of doing it.

The first is to copy and paste the code into a console like, iPython, and then running it.

The second is to place this code in a Python file and then saving it. To run it from the file, go to the directory the file is in and run this command

```
python3 file.py
```

Just replace `file.py` with whatever name you decided to call the file.

3.7.2 Breaking Down the Code

Now that the code has been written let's go through it step-by-step to see what's actually going on.

Reading in the Data

```
geopysc = GeoPyContext(appName="python-ingest", master="local[*]")

# Read the GeoTiff from S3
rdd = get(geopysc, SPATIAL, "s3:///tmp/cropped.tif")
```

Before doing anything when using GeoPySpark, it's best to create a `GeoPyContext` instance. This acts as a wrapper for `SparkContext`, and provides some useful, behind-the-scenes methods for other GeoPySpark functions.

After the creation of `geopysc` we can now read the data. For this example, we will be reading a single GeoTiff that contains only spatial data (hence `SPATIAL`). This will create an instance of *`RasterRDD`* which will allow us to start working with our data.

Collecting the Metadata

```
metadata = rdd.collect_metadata()
```

Before we can begin formatting the data to our desired layout, we must first collect the *`Metadata`* of the entire RDD. The metadata itself will contain the *`TileLayout`* that the data will be formatted to. There are various ways to collect the metadata depending on how you want the layout to look (see *`collect_metadata()`*), but for this example, we will just go with the default parameters.

Tiling the Data

```
# tile the rdd to the layout defined in the metadata
laid_out = rdd.tile_to_layout(metadata)

# reproject the tiled rasters using a ZoomedLayoutScheme
reprojected = laid_out.reproject("EPSG:3857", scheme=ZOOM)
```

With the metadata collected, it is now time to format the data within the RDD to our desired layout. The aptly named, *tile_to_layout()*, method will cut and arrange the rasters in the RDD to the layout within the metadata; giving us a new class instance of *TiledRasterRDD*.

Having this new class will allow us to perform the final steps of our ingest. While the tiles are now in the correct layout, their CRS is not what we want. It would be great if we could make a tile server from our ingested data, but to do that we'll have to change the projection. `reproject()` will be able to help with this. **If you wish to pyramid your data, it must have a ''scheme'' of ''ZOOM'' before the pyramiding takes place**. Read more about why *here*.

Pyramiding the Data

```
# pyramid the TiledRasterRDD to create 12 new TiledRasterRDD
# one for each zoom level
pyramided = reprojected.pyramid(start_zoom=12, end_zoom=1)
```

Now it's time to pyramid! Using our reprojected data, we can create 12 new instances of `TiledRasterRDD`. Each instance represents the data within the RDD at a specific zoom level. **Note**: The `start_zoom` is always the larger number when pyramiding.

Saving the Ingest Locally

```
# Save each TiledRasterRDD locally
for tiled in pyramided:
    write("file:///tmp/python-catalog", "python-ingest", tiled)
```

All that's left to do now is to save it. Since `pyramided` is just a list of `TiledRasterRDD`, we can just loop through it and save each element one at a time.

3.8 Creating a Tile Server From Greyscale Data

Now that we have ingested data, we can use it using a tile server. We will be using the catalog that was created in *Ingesting a Grayscale Image*.

Note: GeoPySpark can create a tile server from a catalog that was created via GeoTrellis!

3.8.1 The Code

Here is the code itself. We will be using `flask` to create a local server and `Pillow` to create our images. For this example, we are working with singleband, grayscale images; so we do not need to worry about color correction.

```
import io
import numpy as np
```

```
from PIL import Image
from flask import Flask, make_response
from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis.catalog import read_value
from geopyspark.geotrellis.constants import SPATIAL


app = Flask(__name__)

@app.route("/<int:zoom>/<int:x>/<int:y>.png")
def tile(x, y, zoom):

    # fetch tile
    tile = read_value(geopycontext,
                      SPATIAL,
                      uri,
                      layer_name,
                      zoom,
                      x,
                      y)

    data = np.int32(tile['data']).reshape(256, 256)

    # display tile
    bio = io.BytesIO()
    im = Image.fromarray(data).resize((256, 256), Image.NEAREST).convert('L')
    im.save(bio, 'PNG')

    response = make_response(bio.getvalue())
    response.headers['Content-Type'] = 'image/png'
    response.headers['Content-Disposition'] = 'filename=%d.png' % 0

    return response


if __name__ == "__main__":
    uri = "file:///tmp/python-catalog/"
    layer_name = "python-ingest"

    geopycontext = GeoPyContext(appName="server-example", master="local[*]")

    app.run()
```

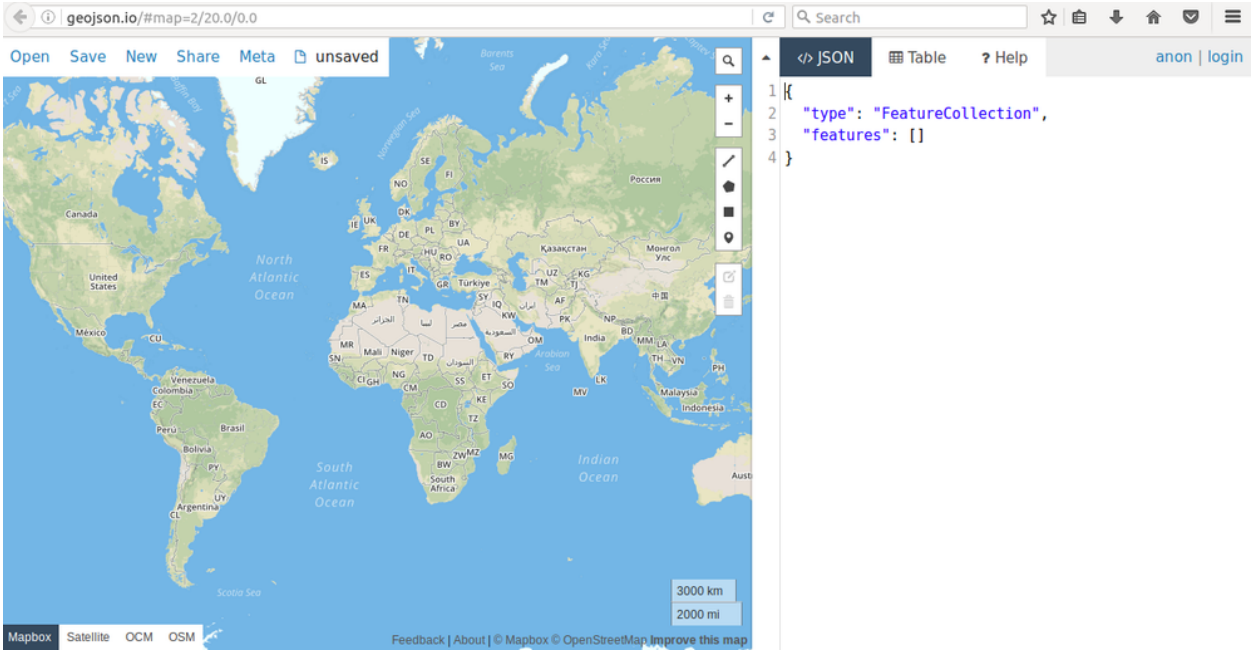
Running the Code

You will want to run this code through the command line. To run it, from the file, go to the directory the file is in and run this command

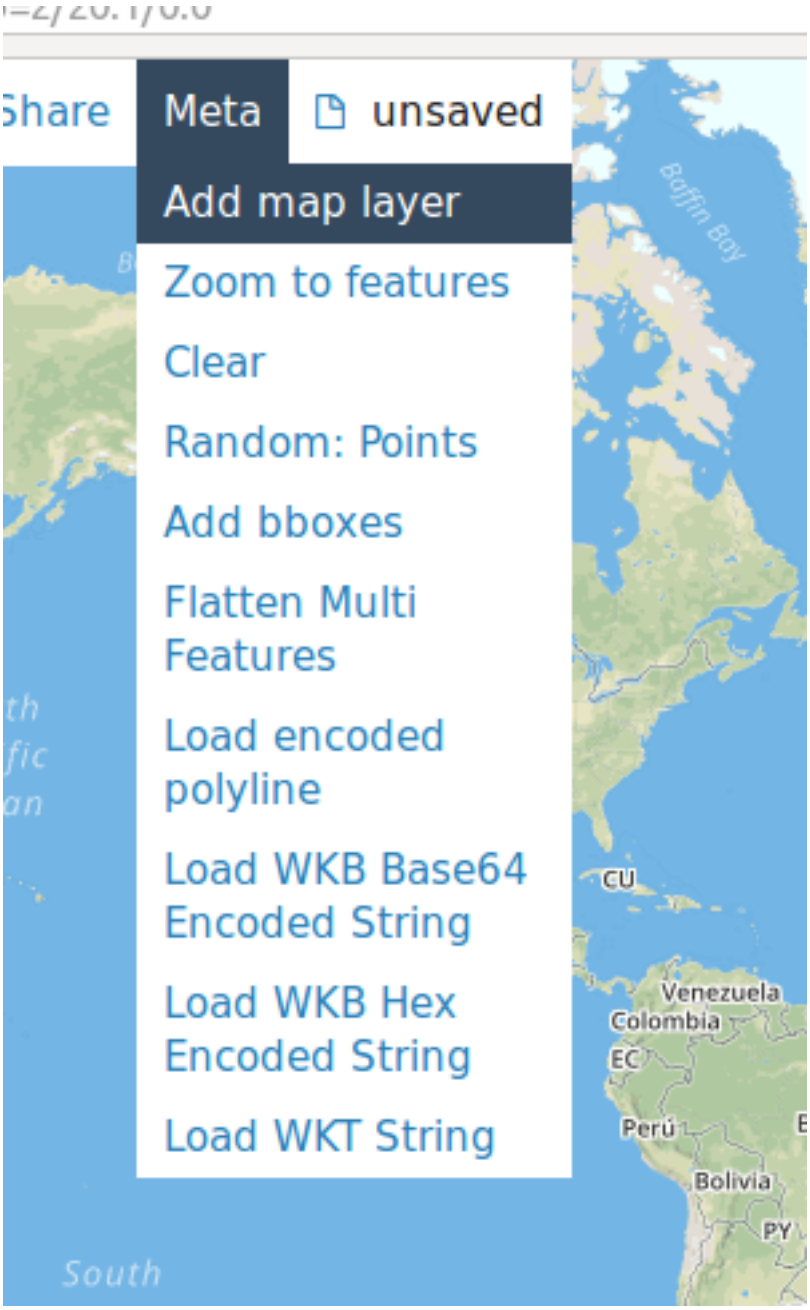
```
python3 file.py
```

Just replace `file.py` with whatever name you decided to call the file.

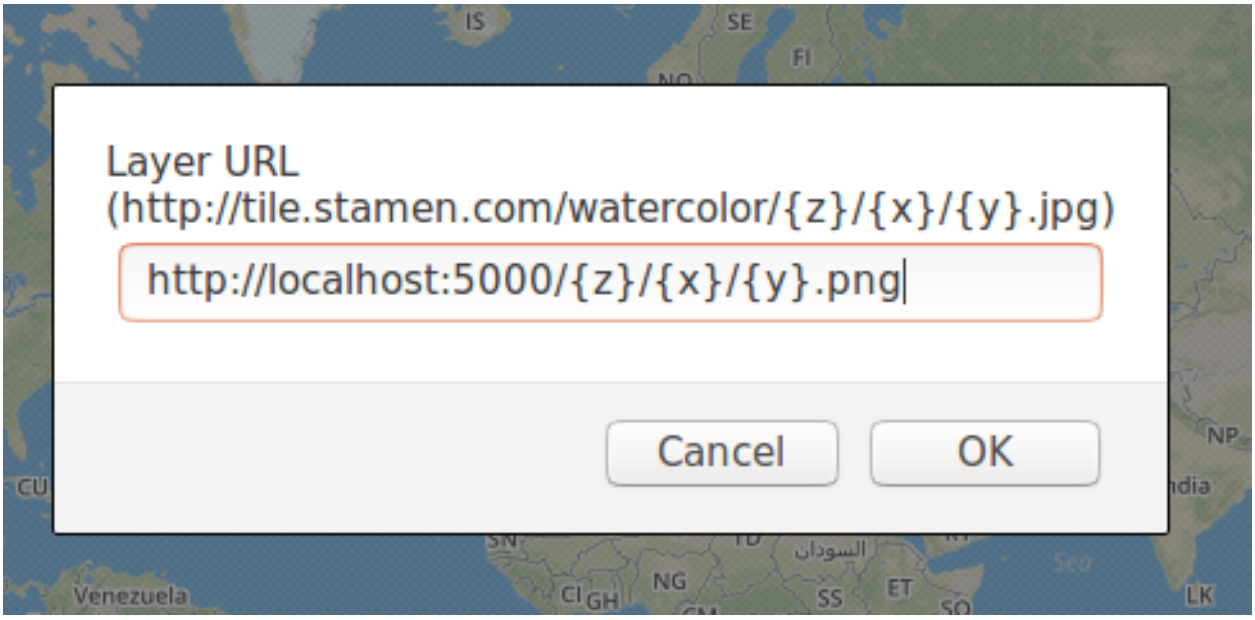
Once it's started, you'll then want to go to a website that allows you to display geo-spatial images from a server. For this example, we'll be using geojson.io, but feel free to use whatever service you want.



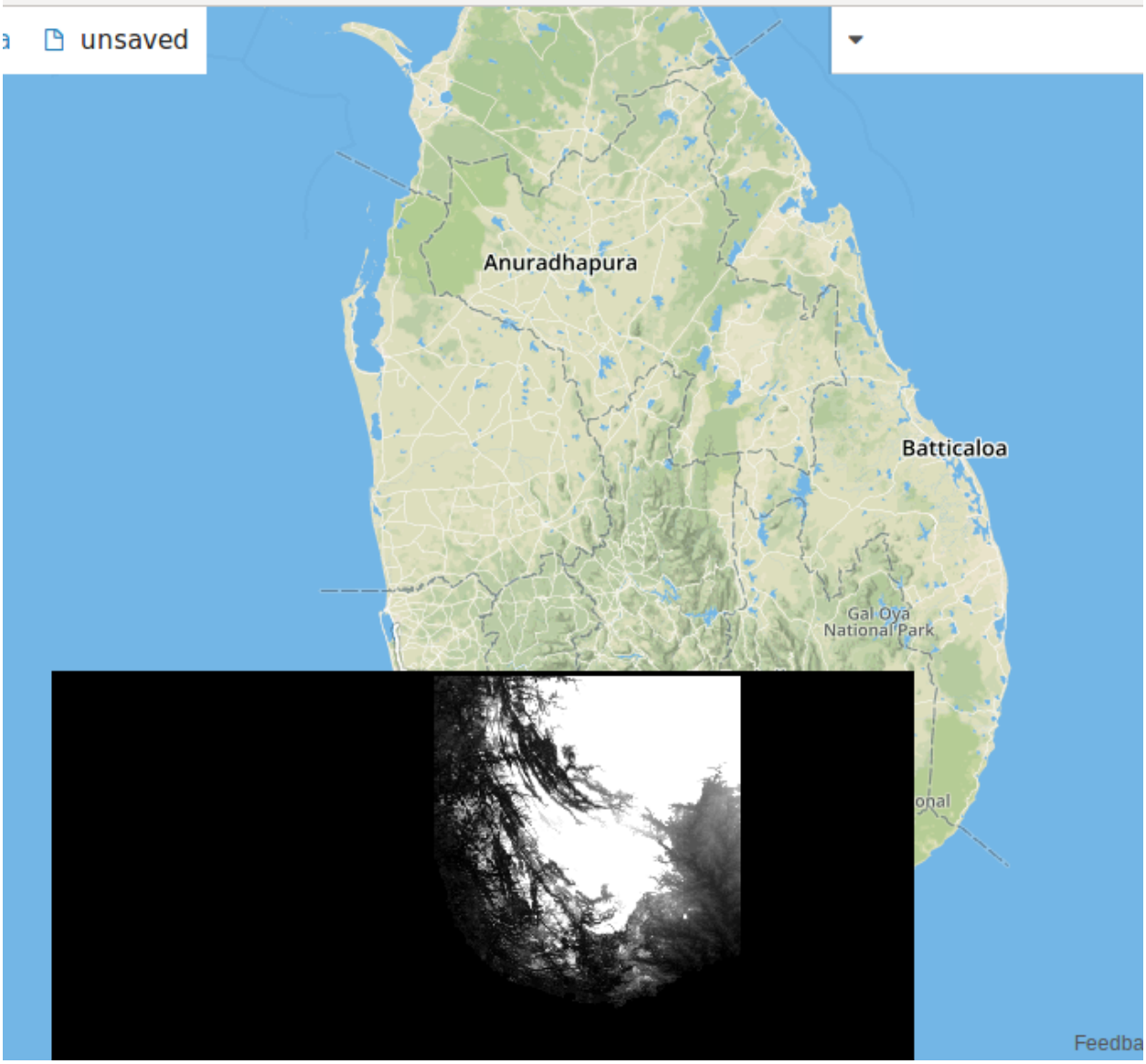
Go to geojson.io, and select the `Meta` option from the tool bar, and then choose the `Add map layer` command.



A pop up will appear where it will ask for the template, layer URL. To get this example to work, please enter the following: `http://localhost:5000/{z}/{x}/{y}.png`.



A second window will appear asking to name the new layer. Pick whatever you want. I tend to use simple names like `a`, `b`, `c`, etc.



Now that everything is setup, it's time to see the image. You'll need to scroll in to Sri Lanka and a black-and-white elevation map should appear. If what you're seeing matches the image above, then the tile server works!

3.8.2 Breaking Down the Code

As with our other examples, let's go through it step-by-step to see what's actually going on. Though, for this example, we'll be starting at the bottom and working our way up.

Setup

```
if __name__ == "__main__":
    uri = "file:///tmp/python-catalog/"
    layer_name = "python-benchmark"

    geopycontext = GeoPyContext(appName="server-example", master="local[*]")
```

```
    app.run()
```

Before getting the tiles, we'll need to setup some constants that will be used. In this case, the `uri`, `layer_name`, and GeoPyContext will remain the same each time a tile is fetched. This is also where `flask` is started via `app.run()`.

Fetching the Tile

```
app = Flask(__name__)

@app.route("/<int:zoom>/<int:x>/<int:y>.png")
def tile(x, y, zoom):

    # fetch tile
    tile = read_value(geopycontext,
                      SPATIAL,
                      uri,
                      layer_name,
                      zoom,
                      x,
                      y)

    data = np.int32(tile['data']).reshape(256, 256)

    # display tile
    bio = io.BytesIO()
    im = Image.fromarray(data).resize((256, 256), Image.NEAREST).convert('L')
    im.save(bio, 'PNG')

    response = make_response(bio.getvalue())
    response.headers['Content-Type'] = 'image/png'
    response.headers['Content-Disposition'] = 'filename=%d.png' % 0

    return response
```

This section of the code is where the tile read from the catalog and made into a PNG which can then be displayed. Because the tiles are stored as a grid within the catalog, giving the `zoom level`, `col`, and `row` of the tile will allow us to retrieve it.

`read_value()` returns a *Raster*, so we take out the underlying `data` and place it into a new NumPy array where the data type is `int32`.

Once we have the NumPy array, we can turn it into an `Image` which we can then turn into a PNG. We turn this PNG into a `flask response`, which allows the tiles themselves to viewed on geojson.io.

3.9 Ingesting a Sentinel Image

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 can prove difficult to work with. This being, `JPEG 2000` (file extension `.jp2`), an image compression format for JPEGs that allow for improved quality and compression ratio.

There are few programs that can work with `jp2`, which can make processing large amounts of them difficult. Because of GeoPySpark, though, we can leverage the tools available to us in Python that can work with `jp2` and use them to

format the sentinel data so that it can be ingested.

Note: This guide goes over how to use `jp2` files with GeoPySpark, the actual ingest process itself is discussed in more detail in *Greyscale Ingest Code Breakdown*.

3.9.1 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, and we will be downloading it straight from there.

The way the data is stored on S3 will not be discussed here, instead, a general overview of the data will be given. We will downloading three different `jp2` that represent the same area and time in different wavelength. These being: Aerosol detection (443 nm), Water vapor (945 nm), and Cirrus (1375 nm). Why these three bands? It's because of the resolution of the image, which determines what bands are represented best. For this example, we will be working at a 60 m resolution; which provides the best representation of the mentioned bands. As for what is in the photos, it is the eastern coast of Corsica taken on January 4th, 2017.

All of the above helps create the following base url for this image set, which we will assign to the `baseurl` variable in the terminal:

```
baseurl="http://sentinel-s2-l1c.s3.amazonaws.com/tiles/32/T/NM/2017/1/4/0/"
```

To download the bands, we just have to `wget` each one, and then move the resulting `jp2` to `/tmp`.

```
wget ${baseurl}B01.jp2 ${baseurl}B09.jp2 ${baseurl}B10.jp2
mv B01.jp2 B09.jp2 B10.jp2 /tmp
```

Now that we have our data, we can now begin the ingest process.

3.9.2 The Code

```
import numpy as np
import rasterio

from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis.constants import SPATIAL, ZOOM
from geopyspark.geotrellis.catalog import write
from geopyspark.geotrellis.rdd import RasterRDD


geopysc = GeoPyContext(appName="sentinel-ingest", master="local[*]")

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

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

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

# saving the max and min values of the tile with
open('/tmp/sentinel_stats.txt', 'w') as f:
    f.writelines([str(data.max()) + "\n", str(data.min())])
```

```
if f.nodata:
    no_data = f.nodata
else:
    no_data = 0

bounds = f.bounds
epsg_code = int(f.crs.to_dict()['init'][5:])

# Creating the RasterRDD
tile = {'data': data, 'no_data_value': no_data}

extent = {'xmin': bounds.left, 'ymin': bounds.bottom, 'xmax': bounds.right, 'ymax':␣
↪bounds.top}
projected_extent = {'extent': extent, 'epsg': epsg_code}

rdd = geopysc.pysc.parallelize([(projected_extent, tile)])
raster_rdd = RasterRDD.from_numpy_rdd(geopysc, SPATIAL, rdd)

metadata = raster_rdd.collect_metadata()
laid_out = raster_rdd.tile_to_layout(metadata)
reprojected = laid_out.reproject("EPSG:3857", scheme=ZOOM)

pyramided = reprojected.pyramid(start_zoom=12, end_zoom=1)

for tiled in pyramided:
    write("file:///tmp/sentinel-catalog", "sentinel-benchmark", tiled)
```

Running the Code

Running the code is simple, and you have two different ways of doing it.

The first is to copy and paste the code into a console like, iPython, and then running it.

The second is to place this code in a python file and then saving it. To run it from the file, go to the directory the file is in and run this command:

```
python3 file.py
```

Just replace `file.py` with whatever name you decided to call the file.

3.9.3 Breaking Down the Code

Let's now see what's going on through the code by going through each step of the process. **Note**: As mentioned in the opening, this section will only cover the reading in and formatting the data steps. For a guide through each ingest step, please see *Greyscale Ingest Code Breakdown*.

The Imports

The one note to make here is:

```
import rasterio
import numpy as np
```

We will need `rasterio` to read in the *jp2*' and `numpy` to format the data so that it can be used with GeoPySpark.

Reading in the JPEG 2000s

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

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

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

`rasterio` being backed by GDAL allows us to read in the `jp2`. Because each image represents a wavelength, there is a order in which they need to be in when they're merged to together into a multiband raster which is represented by `jp2s`. After the reading process, the list of `numpy` arrays will be turned into one array. This represents our mulitband raster.

Saving the Whole Image Stats

```
# saving the max and min values of the tile with
open('/tmp/sentinel_stats.txt', 'w') as f:
    f.writelines([str(data.max()) + "\n", str(data.min())])
```

When we create the tile server for our sentinel images, the data of the `numpy` arrays will need to be converted to the `uint8` data type in order to be represented as a RGB image. In order to do that, though, we will need to normalize each array so that all of the points fall between 0 and 255. This posss a problem, since only a section of the original image is read in and rendered at a time, there is no way of normalizing correctly; as we do not know the entire range of values from the original image. This is why we must save the `max` and `min` values of the whole image in a seperate file to read in later.

Formatting the Data

```
if f.nodata:
    no_data = f.nodata
else:
    no_data = 0

bounds = f.bounds
epsg_code = int(f.crs.to_dict()['init'][5:])

extent = {'xmin': bounds.left, 'ymin': bounds.bottom, 'xmax': bounds.right, 'ymax':␣
↪bounds.top}
projected_extent = {'extent': extent, 'epsg': epsg_code}

rdd = geopysc.pysc.parallelize([(projected_extent, tile)])
raster_rdd = RasterRDD.from_numpy_rdd(geopysc, SPATIAL, rdd)
```

GeoPySpark is a Python binding of GeoTrellis, and because of that, requires the data to be in a certain format. Please see *Core Concepts* to learn what each of these variables represent.

The main take-away from this section of code: if you wish to produce either a `RasterRDD` or `TiledRasterRDD` in Python, then the data **must** be in the correct format.

Ingesting the Data

All that remains now is to ingest the data. These steps can be followed at *Greyscale Ingest Code Breakdown*.

3.10 Creating a Tile Server From Sentinel Data

Now that we have ingested data, we can use it using a tile server. We will be using the catalog that was created in *Ingesting a Sentinel Image*.

Note: This guide will focus on converting the raster into a `Pillow`, RGB image so that it can be used by the tile server. The tile server process itself is discussed in more detail in *Greyscale Tile Server Code Breakdown*.

3.10.1 The Code

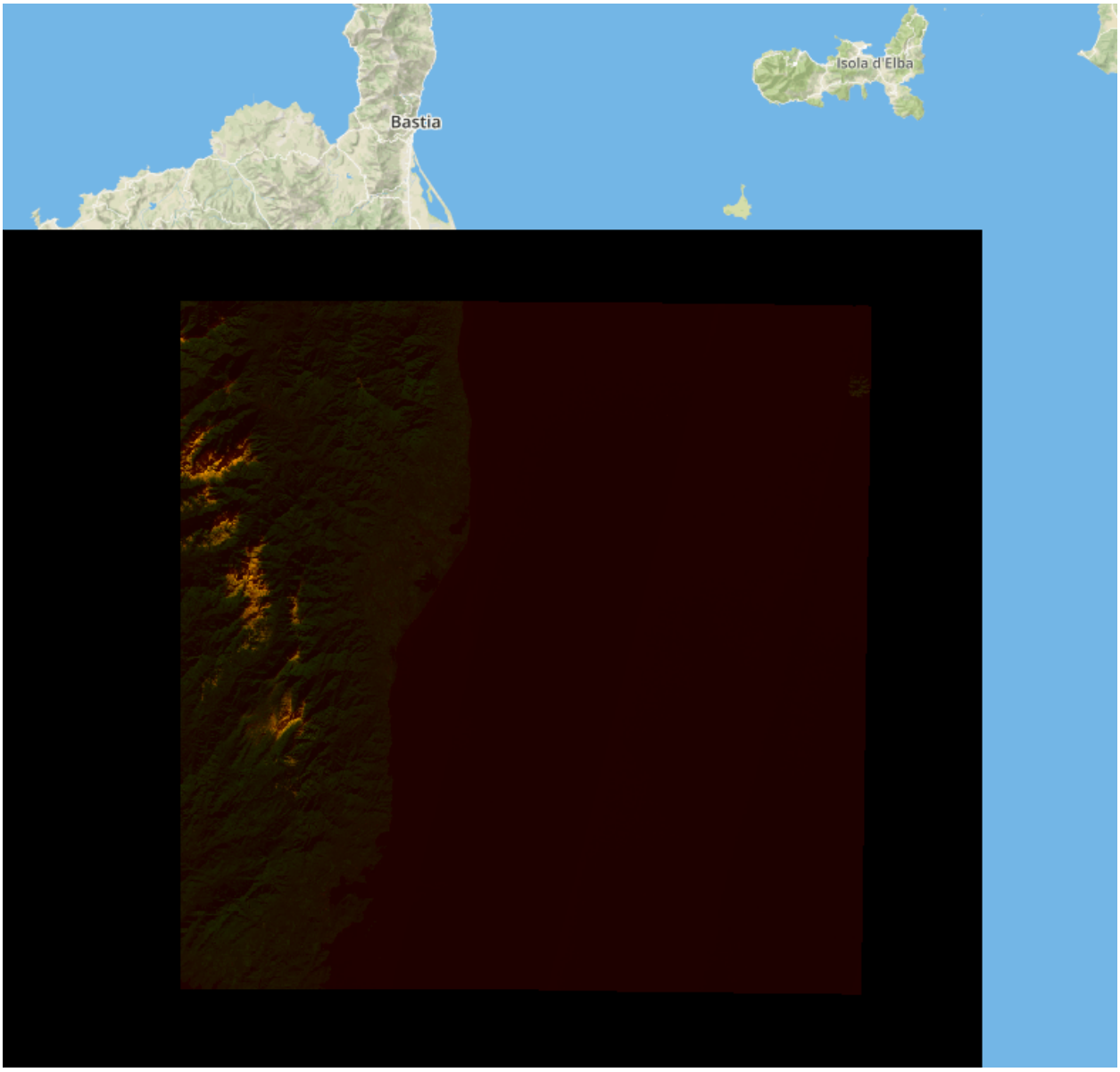
Because we are working with a RGB, a multiband image, we will need to correct the colors for each tile in order for it to displayed correctly.

```
import io
import numpy as np
import rasterio

from flask import Flask, make_response
from PIL import Image

from geopyspark.geopycontext import GeoPyContext
from geopyspark.geotrellis.catalog import read_value
from geopyspark.geotrellis.constants import SPATIAL


# normalize the data so that it falls in the range of 0 - 255
def make_image(arr):
    adjusted = ((arr - whole_min) * 255) / (whole_max - whole_min)
    return Image.fromarray(adjusted.astype('uint8')).resize((256, 256), Image.
↪NEAREST).convert('L')


app = Flask(__name__)

@app.route("/<int:zoom>/<int:x>/<int:y>.png")
def tile(x, y, zoom):
    # fetch tile

    tile = read_value(geopysc, SPATIAL, uri, layer_name, zoom, x, y)
    arr = tile['data']

    bands = arr.shape[0]
    arrs = [np.array(arr[x, :, :]).reshape(256, 256) for x in range(bands)]

    # display tile
    images = [make_image(arr) for arr in arrs]
    image = Image.merge('RGB', images)

    bio = io.BytesIO()
    image.save(bio, 'PNG')
    response = make_response(bio.getvalue())
```

```
    response.headers['Content-Type'] = 'image/png'
    response.headers['Content-Disposition'] = 'filename=%d.png' % 0

    return response


if __name__ == "__main__":
    uri = "file:///tmp/sentinel-catalog"
    layer_name = "sentinel-example"

    geopysc = GeoPyContext(appName="s3-flask", master="local[*]")

    with open('/tmp/sentinel_stats.txt', 'r') as f:
        lines = f.readlines()
        whole_max = int(lines[0])
        whole_min = int(lines[1])

    app.run()
```

Running the Code

Running the tile server is done the same way as in *Greyscale Tile Server Running the Code*. The only difference being the resulting image, of course.



You'll need to scroll over Corsica, and you should see something that matches the above image. If you do, then the server works!

3.10.2 Breaking Down the Code

This next section will go over how to prepare the RGB image to be served. For a more of a general overview of to setup a tile server please see *Greyscale Tile Server Code Breakdown*.

Setup

```
if __name__ == "__main__":
    uri = "file:///tmp/sentinel-catalog"
    layer_name = "sentinel-example"
```

```
    geopysc = GeoPyContext(appName="s3-flask", master="local[*]")

    with open('/tmp/sentinel_stats.txt', 'r') as f:
        lines = f.readlines()
        whole_max = int(lines[0])
        whole_min = int(lines[1])

    app.run()
```

In additon to setting up `uri` and `layer_name`, we will also read in the `max` and `min` values that we saved earlier. These will be used when we normalize a tile.

Preparing the Tile

```
# normalize the data so that it falls in the range of 0 - 255
def make_image(arr):
    adjusted = ((arr - whole_min) * 255) / (whole_max - whole_min)
    return Image.fromarray(adjusted.astype('uint8')).resize((256, 256), Image.
↪NEAREST).convert('L')

app = Flask(__name__)

@app.route("/<int:zoom>/<int:x>/<int:y>.png")
def tile(x, y, zoom):
    # fetch tile

    tile = read_value(geopysc, SPATIAL, uri, layer_name, zoom, x, y)
    arr = tile['data']

    bands = arr.shape[0]
    arrs = [np.array(arr[x, :, :]).reshape(256, 256) for x in range(bands)]

    # display tile
    images = [make_image(arr) for arr in arrs]
    image = Image.merge('RGB', images)
```

Tiles that contain multibands need some work done before they can be served. The `make_image` method takes each band and normalizes it between a range of 0 and 255. We need to do this because `Pillow` expects the data types of arrays to be `uint8`. This is why we need the `whole_max` and the `whole_min` values; as we needed to know the full range of the original values before normalization. Information that would be otherwise impossible to get at this point.

Once normalized, the band is then converted to a greyscale image. This is done for each band in the tile, and once complete, we can then make a RGB `png` file. After this step, the remaining process is no different than if you were working with a singleband tile.

Any details that we not discussed in this document can be found in *Greyscale Tile Server Code Breakdown*.

3.11 geopyspark package

3.11.1 geopyspark

class geopyspark.geopycontext.**AvroRegistry**

Holds the encoding/decoding methods needed to bring a scala RDD to/from Python.

classmethod create_partial_tuple_decoder(*key_type=None*, *value_type=None*)
Creates a partial, tuple decoder function.

Parameters

- **key_type** (*str, optional*) – The type of the key in the tuple.
- **value_type** (*str, optional*) – The type of the value in the tuple.

Returns A partial tuple_decoder function that requires a schema_dict to execute.

classmethod create_partial_tuple_encoder(*key_type=None*, *value_type=None*)
Creates a partial, tuple encoder function.

Parameters

- **key_type** (*str, optional*) – The type of the key in the tuple.
- **value_type** (*str, optional*) – The type of the value in the tuple.

Returns A partial tuple_encoder function that requires a obj to execute.

classmethod tile_decoder(*schema_dict*)
Decodes a TILE into Python.

Parameters **schema_dict** (*dict*) – The dict representation of the AvroSchema.

Returns *Tile*

classmethod tile_encoder(*obj*)
Encodes a TILE to send to Scala.

Parameters **obj** (*dict*) – The dict representation of TILE.

Returns avro_schema_dict (dict)

static tuple_decoder(*schema_dict*, *key_decoder=None*, *value_decoder=None*)
Decodes a tuple into Python.

Parameters

- **schema_dict** (*dict*) – The dict representation of the AvroSchema.
- **key_decoder** (*func, optional*) – The decoding function of the key.
- **value_decoder** (*func, optional*) – The decoding function fo the value.

Returns tuple

static tuple_encoder(*obj*, *key_encoder=None*, *value_encoder=None*)
Encodes a tuple to send to Scala.

Parameters

- **obj** (*tuple*) – The tuple to be encoded.
- **key_encoder** (*func, optional*) – The encoding function of the key.
- **value_encoder** (*func, optional*) – The encoding function fo the value.

Returns avro_schema_dict (dict)

class geopyspark.geopycontext.**AvroSerializer**(*schema*, *decoding_method=None*, *encoding_method=None*)
The serializer used by a RDD to encode/decode values to/from Python.

Parameters

- **schema** (*str*) – The AvroSchema of the RDD.

- **decoding_method** (*func, optional*) – The decocding function for the values within the RDD.
- **encoding_method** (*func, optional*) – The encocding function for the values within the RDD.

schema
str – The AvroSchema of the RDD.

decoding_method
func, optional – The decocding function for the values within the RDD.

encoding_method
func, optional – 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.

schema_dict
The schema values in a dict.

class geopyspark.geopycontext.**GeoPyContext**(*pysc=None*, ***kwargs*)
A wrapper of `SparkContext`. This wrapper provides extra functionality by providing methods that help with sending/recieving information to/from python.

Parameters

- **pysc** (*pypspark.SparkContext, optional*) – An existing `SparkContext`.
- ****kwargs** – `GeoPyContext` can create a `SparkContext` if given its constructing arguments.

Note: If both `pysc` and `kwargs` are set the `pysc` will be used.

---

pysc
pyspark.SparkContext – The wrapped `SparkContext`.

sc
org.apache.spark.SparkContext – The scala `SparkContext` derived from the python one.

Raises `TypeError` – If neither a `SparkContext` or its constructing arguments are given.

Examples

Creating `GeoPyContext` from an existing `SparkContext`.

```
>>> sc = SparkContext(appName="example", master="local[*]")
>>> SparkContext
>>> geopysc = GeoPyContext(sc)
>>> GeoPyContext
```

Creating `GeoPyContext` from the constructing arguments of `SparkContext`.

```
>>> geopysc = GeoPyContext(appName="example", master="local[*]")
>>> GeoPyContext
```

create_python_rdd(*jrdd*, *serializer*)
Creates a Python RDD from a RDD from Scala.

Parameters

- **jrdd** (*org.apache.spark.api.java.JavaRDD*) – The RDD that came from Scala.
- **serializer** (`AvroSerializer` or pyspark.serializers.AutoBatchedSerializer(AvroSerializer)) – An instance of `AvroSerializer` that is either alone, or wrapped by `AutoBatchedSerializer`.

Returns `pyspark.RDD`

create_schema(*key_type*)
Creates an AvroSchema.

Parameters **key_type** (*str*) – The type of the `K` in the tuple, `(K, V)` in the RDD.

Returns An AvroSchema for the types within the RDD.

static map_key_input(*key_type*, *is_boundable*)
Gets the mapped GeoTrellis type from the *key_type*.

Parameters

- **key_type** (*str*) – The type of the `K` in the tuple, `(K, V)` in the RDD.
- **is_boundable** (*bool*) – Is `K` boundable.

Returns The corresponding GeoTrellis type.

3.12 geopyspark.geotrellis package

This subpackage contains the code that reads, writes, and processes data using GeoTrellis.

class geopyspark.geotrellis.**Bounds**(*minKey*, *maxKey*)
Represents the grid that covers the area of the rasters in a RDD on a grid.

Parameters

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

Returns *Bounds*

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.**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.**LayoutDefinition**(*extent*, *tileLayout*)
Describes the layout of the rasters within a RDD and how they are projected.

Parameters

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

Returns *LayoutDefinition*

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.**Metadata**(*bounds*, *crs*, *cell_type*, *extent*, *layout_definition*)
Information of the values within a RasterRDD or TiledRasterRDD. 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*) – 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.

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 `RasterRDD` or `TiledRasterRDD` instance that is in `dict` form.

Returns *Metadata*

to_dict()
Converts this instance to a `dict`.

Returns `dict`

class geopyspark.geotrellis.**TileLayout**(*layoutCols*, *layoutRows*, *tileCols*, *tileRows*)
Describes the grid in which the rasters within a RDD 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.

Returns *TileLayout*

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

3.12.1 geopyspark.geotrellis.catalog module

Methods for reading, querying, and saving tile layers to and from GeoTrellis Catalogs.

geopyspark.geotrellis.catalog.**get_layer_ids**(*geopysc*, *uri*, *options=None*, ***kwargs*)
Returns a list of all of the layer ids in the selected catalog as dicts that contain the name and zoom of a given layer.

Parameters

- **geopysc** (*geopyspark.GeoPyContext*) – The `GeoPyContext` being used this session.
- **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.
- **options** (*dict, optional*) – Additional parameters for reading the layer for specific backends. The dictionary is only used for Cassandra and HBase, no other backend requires this to be set.

- ****kwargs** – The optional parameters can also be set as keywords arguments. The keywords must be in camel case. If both options and keywords are set, then the options will be used.

Returns

[layerIds]

Where `layerIds` is a `dict` with the following fields:

- **name** (str): The name of the layer
- **zoom** (int): The zoom level of the given layer.

`geopyspark.geotrellis.catalog.`**`query`**(*geopysc*, *rdd_type*, *uri*, *layer_name*, *layer_zoom*, *intersects*, *time_intervals=None*, *proj_query=None*, *options=None*, *numPartitions=None*, ***kwargs*)

Queries a single, zoom layer from a GeoTrellis catalog given spatial and/or time parameters. Unlike read, this method will only return part of the layer that intersects the specified region.

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

- **geopysc** (`GeoPyContext`) – The GeoPyContext being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: `SPATIAL` and `SPACETIME`. Note: All of the GeoTiffs must have the same saptial type.
- **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*) – The zoom level of the layer that is to be querried.
- **intersects** (str or Polygon or `Extent`) – The desired spatial area to be returned. Can either be a string, a shapely Polygon, or an instance of `Extent`. If the value is a string, it must be the WKT string, geometry format.

  The types of Polygons 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.

- **time_intervals** (*list, optional*) – A list of strings that time intervals to query. The strings must be in a valid date-time format. 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.

- **options** (*dict, optional*) – Additional parameters for querying the tile for specific backends. The dictioanry is only used for `Cassandra` and `HBase`, no other backend requires this to be set.
- **numPartitions** (*int, optional*) – Sets RDD partition count when reading from catalog.
- ****kwargs** – The optional parameters can also be set as keywords arguements. The keywords must be in camel case. If both options and keywords are set, then the options will be used.

Returns *TiledRasterRDD*

geopyspark.geotrellis.catalog.**read**(*geopysc*, *rdd_type*, *uri*, *layer_name*, *layer_zoom*, *options=None*, *numPartitions=None*, ***kwargs*)

Reads a single, zoom layer from a GeoTrellis catalog.

Note: This will read the entire layer. If only part of the layer is needed, use *query()* instead.

Parameters

- **geopysc** (`GeoPyContext`) – The GeoPyContext being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: `SPATIAL` and `SPACETIME`.
- **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.
- **options** (*dict, optional*) – Additional parameters for reading the layer for specific backends. The dictionary is only used for `Cassandra` and `HBase`, no other backend requires this to be set.
- **numPartitions** (*int, optional*) – Sets RDD partition count when reading from catalog.
- ****kwargs** – The optional parameters can also be set as keywords arguments. The keywords must be in camel case. If both options and keywords are set, then the options will be used.

Returns *TiledRasterRDD*

geopyspark.geotrellis.catalog.**read_layer_metadata**(*geopysc*, *rdd_type*, *uri*, *layer_name*, *layer_zoom*, *options=None*, ***kwargs*)

Reads the metadata from a saved layer without reading in the whole layer.

Parameters

- **geopysc** (*geopyspark.GeoPyContext*) – The `GeoPyContext` being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: `SPATIAL` and `SPACETIME`.
- **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.
- **options** (*dict, optional*) – Additional parameters for reading the layer for specific backends. The dictionary is only used for `Cassandra` and `HBase`, no other backend requires this to be set.
- **numPartitions** (*int, optional*) – Sets RDD partition count when reading from catalog.
- ****kwargs** – The optional parameters can also be set as keywords arguments. The keywords must be in camel case. If both options and keywords are set, then the options will be used.

Returns *Metadata*

geopyspark.geotrellis.catalog.**read_value**(*geopysc*, *rdd_type*, *uri*, *layer_name*, *layer_zoom*, *col*, *row*, *zdt=None*, *options=None*, ***kwargs*)

Reads a single tile from a GeoTrellis catalog. Unlike other functions in this module, this will not return a `TiledRasterRDD`, but rather a GeoPySpark formatted raster. This is the function to use when creating a tile server.

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

Parameters

- **geopysc** (*geopyspark.GeoPyContext*) – The `GeoPyContext` being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: `SPATIAL` and `SPACETIME`.
- **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** (*str*) – The Zone-Date-Time string of the tile. The string must be in a valid date-time format. This parameter is only used when querying spatial-temporal data. The default value is, None. If None, then only the spatial area will be queried.
- **options** (*dict, optional*) – Additional parameters for reading the tile for specific backends. The dictionary is only used for `Cassandra` and `HBase`, no other backend requires this to be set.
- ****kwargs** – The optional parameters can also be set as keywords arguments. The keywords must be in camel case. If both options and keywords are set, then the options will be used.

Returns *Raster* or `None`

geopyspark.geotrellis.catalog.**write**(*uri*, *layer_name*, *tiled_raster_rdd*, *index_strategy='zorder'*, *time_unit=None*, *options=None*, ***kwargs*)

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_rdd** (*TiledRasterRDD*) – The TiledRasterRDD to be saved.
- **index_strategy** (*str*) – The method used to orginize the saved data. Depending on the type of data within the layer, only certain methods are available. The default method used is, ZORDER.
- **time_unit** (*str, optional*) – Which time unit should be used when saving spatial-temporal data. 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.
- **options** (*dict, optional*) – Additional parameters for writing the layer for specific backends. The dictioanry is only used for Cassandra and HBase, no other backend requires this to be set.
- ****kwargs** – The optional parameters can also be set as keywords arguements. The keywords must be in camel case. If both options and keywords are set, then the options will be used.

3.12.2 geopyspark.geotrellis.constants module

Constants that are used by geopyspark.geotrellis classes, methods, and functions.

geopyspark.geotrellis.constants.**ANNULUS = 'annulus'**
Neighborhood type.

geopyspark.geotrellis.constants.**ASPECT = 'Aspect'**
Focal operation type.

geopyspark.geotrellis.constants.**AVERAGE = 'Average'**
A resampling method.

geopyspark.geotrellis.constants.**BILINEAR = 'Bilinear'**
A resampling method.

geopyspark.geotrellis.constants.**BLUE_TO_ORANGE = 'BlueToOrange'**
A ColorRamp.

geopyspark.geotrellis.constants.**BLUE_TO_RED = 'BlueToRed'**
A ColorRamp.

geopyspark.geotrellis.constants.**BOOL = 'bool'**
Represented Byte Cells with constant NoData values.

geopyspark.geotrellis.constants.**BOOLRAW = 'boolraw'**
Represented Byte Cells.

geopyspark.geotrellis.constants.**CELL_TYPES = ['boolraw', 'int8raw', 'uint8raw', 'int16raw',**
A ColorRamp.

geopyspark.geotrellis.constants.**CIRCLE = 'circle'**
Focal operation type.

geopyspark.geotrellis.constants.**CLASSIFICATION_BOLD_LAND_USE = 'ClassificationBoldLandUse'**
A ColorRamp.

geopyspark.geotrellis.constants.**COOLWARM = 'coolwarm'**
A ColorRamp.

geopyspark.geotrellis.constants.**CUBICCONVOLUTION = 'CubicConvolution'**
A resampling method.

geopyspark.geotrellis.constants.**CUBICSPLINE = 'CubicSpline'**
A resampling method.

geopyspark.geotrellis.constants.**DAYS = 'days'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**EXACT = 'Exact'**
Represented Bit Cells.

geopyspark.geotrellis.constants.**FLOAT = 'float'**
A key indexing method. Works for RDD that contain both *SpatialKey* and *SpaceTimeKey*.

geopyspark.geotrellis.constants.**FLOAT32 = 'float32'**
Represented Double Cells with constant NoData values.

geopyspark.geotrellis.constants.**FLOAT32RAW = 'float32raw'**
Represented Double Cells.

geopyspark.geotrellis.constants.**FLOAT32UD = 'float32ud'**
Represented Double Cells with user defined NoData values.

geopyspark.geotrellis.constants.**FLOAT64 = 'float64'**
Represented Byte Cells with user defined NoData values.

geopyspark.geotrellis.constants.**FLOAT64RAW = 'float64raw'**
Represented Bit Cells.

geopyspark.geotrellis.constants.**GREATERTHAN = 'GreaterThan'**
A classification strategy.

geopyspark.geotrellis.constants.**GREATERTHANOREQUALTO = 'GreaterThanOrEqualTo'**
A classification strategy.

geopyspark.geotrellis.constants.**GREEN_TO_RED_ORANGE = 'GreenToRedOrange'**
A ColorRamp.

geopyspark.geotrellis.constants.**HEATMAP_BLUE_TO_YELLOW_TO_RED_SPECTRUM = 'HeatmapBlueToYell**
A ColorRamp.

geopyspark.geotrellis.constants.**HEATMAP_DARK_RED_TO_YELLOW_WHITE = 'HeatmapDarkRedToYellowW**
A ColorRamp.

geopyspark.geotrellis.constants.**HEATMAP_LIGHT_PURPLE_TO_DARK_PURPLE_TO_WHITE = 'HeatmapLigh**
A ColorRamp.

geopyspark.geotrellis.constants.**HEATMAP_YELLOW_TO_RED = 'HeatmapYellowToRed'**
A ColorRamp.

geopyspark.geotrellis.constants.**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.

geopyspark.geotrellis.constants.**HOT = 'hot'**
A ColorRamp.

geopyspark.geotrellis.constants.**HOURS = 'hours'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**INFERNO = 'inferno'**
A ColorRamp.

geopyspark.geotrellis.constants.**INT16 = 'int16'**
Represented UShort Cells with constant NoData values.

geopyspark.geotrellis.constants.**INT16RAW = 'int16raw'**
Represented UShort Cells.

geopyspark.geotrellis.constants.**INT16UD = 'int16ud'**
Represented UShort Cells with user defined NoData values.

geopyspark.geotrellis.constants.**INT32 = 'int32'**
Represented Float Cells with constant NoData values.

geopyspark.geotrellis.constants.**INT32RAW = 'int32raw'**
Represented Float Cells.

geopyspark.geotrellis.constants.**INT32UD = 'int32ud'**
Represented Float Cells with user defined NoData values.

geopyspark.geotrellis.constants.**INT8 = 'int8'**
Represented UByte Cells with constant NoData values.

geopyspark.geotrellis.constants.**INT8RAW = 'int8raw'**
Represented UByte Cells.

geopyspark.geotrellis.constants.**INT8UD = 'int8ud'**
Represented UByte Cells with user defined NoData values.

geopyspark.geotrellis.constants.**LANCZOS = 'Lanczos'**
A resampling method.

geopyspark.geotrellis.constants.**LESSTHAN = 'LessThan'**
A classification strategy.

geopyspark.geotrellis.constants.**LESSTHANOREQUALTO = 'LessThanOrEqualTo'**
A classification strategy.

geopyspark.geotrellis.constants.**LIGHT_TO_DARK_GREEN = 'LightToDarkGreen'**
A ColorRamp.

geopyspark.geotrellis.constants.**LIGHT_TO_DARK_SUNSET = 'LightToDarkSunset'**
A ColorRamp.

geopyspark.geotrellis.constants.**LIGHT_YELLOW_TO_ORANGE = 'LightYellowToOrange'**
A ColorRamp.

geopyspark.geotrellis.constants.**MAGMA = 'magma'**
A ColorRamp.

geopyspark.geotrellis.constants.**MAX = 'Max'**
A resampling method.

geopyspark.geotrellis.constants.**MEAN = 'Mean'**
Focal operation type

geopyspark.geotrellis.constants.**MEDIAN = 'Median'**
A resampling method.

geopyspark.geotrellis.constants.**MILLISECONDS = 'millis'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**MINUTES = 'minutes'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**MODE = 'Mode'**
A resampling method.

geopyspark.geotrellis.constants.**MONTHS = 'months'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**NEARESTNEIGHBOR = 'NearestNeighbor'**
A resampling method.

geopyspark.geotrellis.constants.**NEIGHBORHOODS = ['annulus', 'nesw', 'square', 'wedge', 'ci**
The NoData value for ints in GeoTrellis.

geopyspark.geotrellis.constants.**NESW = 'nesw'**
Neighborhood type.

geopyspark.geotrellis.constants.**NODATAINT = -2147483648**
A classification strategy.

geopyspark.geotrellis.constants.**PLASMA = 'plasma'**
A ColorRamp.

geopyspark.geotrellis.constants.**RESAMPLE_METHODS = ['NearestNeighbor', 'Bilinear', 'CubicC**
Layout scheme to match resolution of the closest level of TMS pyramid.

geopyspark.geotrellis.constants.**ROWMAJOR = 'rowmajor'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**SECONDS = 'seconds'**
A time unit used with ZORDER.

geopyspark.geotrellis.constants.**SLOPE = 'Slope'**
Focal operation type.

geopyspark.geotrellis.constants.**SPACETIME = 'spacetime'**
Indicates the type value that needs to be serialized/deserialized. Both singleband and multiband GeoTiffs are referred to as this.

geopyspark.geotrellis.constants.**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`.

geopyspark.geotrellis.constants.**SQUARE = 'square'**
Neighborhood type.

geopyspark.geotrellis.constants.**SUM = 'Sum'**
Focal operation type.

geopyspark.geotrellis.constants.**TILE = 'Tile'**
A resampling method.

geopyspark.geotrellis.constants.**UINT16 = 'uint16'**
Represented Int Cells with constant NoData values.

geopyspark.geotrellis.constants.**UINT16RAW = 'uint16raw'**
Represented Int Cells.

geopyspark.geotrellis.constants.**UINT16UD = 'uint16ud'**
Representes Int Cells with user defined NoData values.

geopyspark.geotrellis.constants.**UINT8 = 'uint8'**
Representes Short Cells with constant NoData values.

geopyspark.geotrellis.constants.**UINT8RAW = 'uint8raw'**
Representes Short Cells.

geopyspark.geotrellis.constants.**UINT8UD = 'uint8ud'**
Representes Short Cells with user defined NoData values.

geopyspark.geotrellis.constants.**VIRIDIS = 'viridis'**
A ColorRamp.

geopyspark.geotrellis.constants.**WEDGE = 'wedge'**
Neighborhood type.

geopyspark.geotrellis.constants.**YEARS = 'years'**
Neighborhood type.

geopyspark.geotrellis.constants.**ZOOM = 'zoom'**
Layout scheme to match resolution of source rasters.

geopyspark.geotrellis.constants.**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.

3.12.3 geopyspark.geotrellis.geotiff_rdd module

This module contains functions that create `RasterRDD` from files.

geopyspark.geotrellis.geotiff_rdd.**get**(*geopysc*, *rdd_type*, *uri*, *options=None*, ***kwargs*)
Creates a `RasterRDD` from GeoTiffs that are located on the local file system, `HDFS`, or `S3`.

Parameters

- **geopysc** (*geopyspark.GeoPyContext*) – The `GeoPyContext` being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: `SPATIAL` and `SPACETIME`.

  Note: All of the GeoTiffs must have the same saptial type.

- **uri** (*str*) – The path to a given file/directory.
- **options** (*dict, optional*) – A dictionary of different options that are used when creating the RDD. This defaults to `None`. If `None`, then the RDD will be created using the default options for the given backend in GeoTrellis.

  Note: Key values in the `dict` should be in camel case, as this is the style that is used in Scala.

  These are the options when using the local file system or `HDFS`:

- **crs (str, optional): The CRS that the output tiles should be** in. The CRS must be in the well-known name format. If `None`, then the CRS that the tiles were originally in will be used.
- **timeTag (str, optional): The name of the tiff tag that contains** the time stamp for the tile. If `None`, then the default value is: `TIFFTAG_DATETIME`.
- **timeFormat (str, optional): The pattern of the time stamp for** java.time.format.DateTimeFormatter to parse. If `None`, then the default value is: `yyyy:MM:dd HH:mm:ss`.
- **maxTileSize (int, optional): The max size of each tile in the** resulting RDD. If the size is smaller than a read in tile, then that tile will be broken into tiles of the specified size. If `None`, then the whole tile will be read in.
- **numPartitions (int, optional): The number of repartitions Spark** will make when the data is repartitioned. If `None`, then the data will not be repartitioned.
- **chunkSize (int, optional): How many bytes of the file should be** read in at a time. If None, then files will be read in 65536 byte chunks.

`S3` has the above options in addition to this:

- **s3Client (str, optional): Which `S3Cleint` to use when reading** GeoTiffs. There are currently two options: `default` and `mock`. If `None`, `defualt` is used.

  Note: `mock` should only be used in unit tests and debugging.

- ****kwargs*** – Option parameters can also be entered as keyword arguements.

Note: Defining both `options` and `kwargs` will cause the `kwargs` to be ignored in favor of `options`.

Returns *RasterRDD*

3.12.4 geopyspark.geotrellis.neighborhoods 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.neighborhoods.**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".

class geopyspark.geotrellis.neighborhoods.**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.neighborhoods.**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.neighborhoods.**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".

3.12.5 geopyspark.geotrellis.rdd module

This module contains the `RasterRDD` and the `TiledRasterRDD` 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.rdd.**CachableRDD**
Base class for class that wraps a Scala RDD instance through a py4j reference.

geopysc
`GeoPyContext` – The `GeoPyContext` being used this session.

srdd
py4j.java_gateway.JavaObject – The coresponding Scala RDD class.

cache()
Persist this RDD with the default storage level (C{MEMORY_ONLY}).

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 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.rdd.**RasterRDD**(*geopysc*, *rdd_type*, *srdd*)
A wrapper of a RDD that contains GeoTrellis rasters.

Represents a RDD that contains `(K, V)`. Where `K` is either *ProjectedExtent* or *TemporalProjectedExtent* depending on the `rdd_type` of the RDD, and `V` being a *Raster*.

The data held within the RDD has not been tiled. Meaning the data has yet to be modified to fit a certain layout. See *RasterRDD* for more information.

Parameters

- **geopysc** (`GeoPyContext`) – The `GeoPyContext` being used this session.

- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: SPATIAL and SPACETIME.
- **srdd** (*py4j.java_gateway.JavaObject*) – The coresponding Scala class. This is what allows RasterRDD to access the various Scala methods.

geopysc
GeoPyContext – The GeoPyContext being used this session.

rdd_type
str – What the spatial type of the geotiffs are. This is represented by the constants: SPATIAL and SPACETIME.

srdd
py4j.java_gateway.JavaObject – The coresponding Scala class. This is what allows RasterRDD to access the various Scala methods.

cache()
Persist this RDD with the default storage level (C{MEMORY_ONLY}).

collect_metadata(*extent=None*, *layout=None*, *crs=None*, *tile_size=256*)
Iterate over RDD records and generates layer metadata desribing the contained rasters.

Parameters

- **extent** (*Extent*, optional) – Specify layout extent, must also specify layout.
- **layout** (*TileLayout*, optional) – Specify tile layout, must also specify extent.
- **crs** (*str or int, optional*) – Ignore CRS from records and use given one instead.
- **tile_size** (*int, optional*) – Pixel dimensions of each tile, if not using layout.

Note: extent and layout must both be defined if they are to be used.

Returns *Metadata*

Raises TypeError – If either extent and layout is not defined but the other is.

convert_data_type(*new_type*)
Converts the underlying, raster values to a new CellType.

Parameters **new_type** (*str*) – The string representation of the CellType to convert to. It is represented by a constant such as INT16, FLOAT64UD, etc.

Returns *RasterRDD*

Raises ValueError – When an unsupported cell type is entered.

cut_tiles(*layer_metadata*, *resample_method='NearestNeighbor'*)
Cut tiles to layout. May result in duplicate keys.

Parameters

- **layer_metadata** (*Metadata*) – The Metadata of the RasterRDD instance.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants:

NEARESTNEIGHBOR, BILINEAR, CUBICCONVOLUTION, LANCZOS, AVERAGE, MODE, MEDIAN, MAX, and MIN. If none is specified, then NEARESTNEIGHBOR is used.

Returns *TiledRasterRDD*

classmethod from_numpy_rdd(*geopysc*, *rdd_type*, *numpy_rdd*)

Create a RasterRDD from a numpy RDD.

Parameters

- **geopysc** (GeoPyContext) – The GeoPyContext being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: SPATIAL and SPACETIME.
- **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 *RasterRDD*

get_min_max()

Returns the maximum and minimum values of all of the rasters in the RDD.

Returns (float, float)

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*, *boundary_strategy='LessThanOrEqualTo'*, *replace_nodata_with=None*)

Changes the cell values of a raster based on how the data is broken up.

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.
- **boundary_strategy** (*str, optional*) – How the cells should be classified along the breaks. This is represented by the following constants: GREATERTHAN, GREATERTHANOREQUALTO, LESSTHAN, LESSTHANOREQUALTO, and EXACT. If unspecified, then LESSTHANOREQUALTO will be used.
- **replace_nodata_with** (*data_type, 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: NoData symbolizes a different value depending on if data_type is int or float. For int, the constant NODATAINT can be used which represents the NoData value for int in GeoTrellis. For float, float('nan') is used to represent NoData.

Returns *RasterRDD*

reproject(*target_crs*, *resample_method='NearestNeighbor'*)
Reproject every individual raster to `target_crs`, does not sample past tile boundary

Parameters

- **target_crs** (*str or int*) – The CRS to reproject to. Can either be the EPSG code, well-known name, or a PROJ.4 projection string.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants: `NEARESTNEIGHBOR`, `BILINEAR`, `CUBICCONVOLUTION`, `LANCZOS`, `AVERAGE`, `MODE`, `MEDIAN`, `MAX`, and `MIN`. If none is specified, then `NEARESTNEIGHBOR` is used.

Returns *RasterRDD*

tile_to_layout(*layer_metadata*, *resample_method='NearestNeighbor'*)
Cut tiles to layout and merge overlapping tiles. This will produce unique keys.

Parameters

- **layer_metadata** (*Metadata*) – The `Metadata` of the `RasterRDD` instance.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants: `NEARESTNEIGHBOR`, `BILINEAR`, `CUBICCONVOLUTION`, `LANCZOS`, `AVERAGE`, `MODE`, `MEDIAN`, `MAX`, and `MIN`. If none is specified, then `NEARESTNEIGHBOR` is used.

Returns *TiledRasterRDD*

to_numpy_rdd()
Converts a `RasterRDD` 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 `pyspark.RDD`

to_tiled_layer(*extent=None*, *layout=None*, *crs=None*, *tile_size=256*, *resample_method='NearestNeighbor'*)
Converts this `RasterRDD` to a `TiledRasterRDD`.

This method combines *collect_metadata()* and *tile_to_layout()* into one step.

Parameters

- **extent** (*Extent*, optional) – Specify layout extent, must also specify layout.
- **layout** (*TileLayout*, optional) – Specify tile layout, must also specify `extent`.
- **crs** (*str or int, optional*) – Ignore CRS from records and use given one instead.
- **tile_size** (*int, optional*) – Pixel dimensions of each tile, if not using layout.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants:

NEARESTNEIGHBOR, BILINEAR, CUBICCONVOLUTION, LANCZOS, AVERAGE, MODE, MEDIAN, MAX, and MIN. If none is specified, then NEARESTNEIGHBOR is used.

Note: extent and layout must both be defined if they are to be used.

Returns *TiledRasterRDD*

unpersist()
Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

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.rdd.**TiledRasterRDD**(*geopysc*, *rdd_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 rdd_type of the RDD, and V being a *Raster*.

The data held within the RDD is tiled. This means that the rasters have been modified to fit a larger layout. For more information, see *TiledRasterRDD*.

Parameters

- **geopysc** (GeoPyContext) – The GeoPyContext being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: SPATIAL and SPACETIME.
- **srdd** (*py4j.java_gateway.JavaObject*) – The coresponding Scala class. This is what allows TiledRasterRDD to access the various Scala methods.

geopysc
GeoPyContext – The GeoPyContext being used this session.

rdd_type
str – What the spatial type of the geotiffs are. This is represented by the constants: SPATIAL` and ``SPACETIME.

srdd
py4j.java_gateway.JavaObject – The coresponding Scala class. This is what allows RasterRDD to access the various Scala methods.

cache()
Persist this RDD with the default storage level (C{MEMORY_ONLY}).

convert_data_type(*new_type*)
Converts the underlying, raster values to a new CellType.

Parameters **new_type** (*str*) – The string representation of the CellType to convert to. It is represented by a constant such as INT16, FLOAT64UD, etc.

Returns *TiledRasterRDD*

cost_distance(*geometries*, *max_distance*)
Performs cost distance of a TileLayer.

Parameters

- **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, float*) – The maximum cost that a path may reach before the operation. stops. This value can be an `int` or `float`.

Returns *TiledRasterRDD*

classmethod euclidean_distance(*geopysc*, *geometry*, *source_crs*, *zoom*, *cellType='float64'*)

Calculates the Euclidean distance of a Shapely geometry.

Parameters

- **geopysc** (`GeoPyContext`) – The `GeoPyContext` being used this session.
- **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.

Note: This function may run very slowly for polygonal inputs if they cover many cells of the output raster.

Returns `RDD`

focal(*operation*, *neighborhood=None*, *param_1=None*, *param_2=None*, *param_3=None*)

Performs the given focal operation on the layers contained in the RDD.

Parameters

- **operation** (*str*) – The focal operation. Represented by constants: `SUM`, `MIN`, `MAX`, `MEAN`, `MEDIAN`, `MODE`, `STANDARDDEVIATION`, `ASPECT`, and `SLOPE`.
- **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 the constants: `ANNULUS`, `NEWS`, `SQUARE`, `WEDGE`, and `CIRCLE`. Defaults to `None`.
- **param_1** (*int or float, optional*) – If using `SLOPE`, then this is the zFactor, else it is 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`.

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 either `SLOPE` or `ASPECT`.

Returns *TiledRasterRDD*

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 `SLOPE` or `ASPECT`.

classmethod from_numpy_rdd(*geopysc*, *rdd_type*, *numpy_rdd*, *metadata*)
Create a `TiledRasterRDD` from a numpy RDD.

Parameters

- **geopysc** (`GeoPyContext`) – The `GeoPyContext` being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: `SPATIAL` and `SPACETIME`.
- **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 `TiledRasterRDD` instance.

Returns *TiledRasterRDD*

get_histogram()
Returns an array of Java histogram objects, one for each band of the raster.

Parameters **None** –

Returns An array of Java objects containing the histograms of each band

get_min_max()
Returns the maximum and minimum values of all of the rasters in the RDD.

Returns `(float, float)`

get_quantile_breaks(*num_breaks*)
Returns quantile breaks for this RDD.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns `[float]`

get_quantile_breaks_exact_int(*num_breaks*)
Returns quantile breaks for this RDD. This version uses the `FastMapHistogram`, which counts exact integer values. If your RDD has too many values, this can cause memory errors.

Parameters **num_breaks** (*int*) – The number of breaks to return.

Returns `[int]`

is_floating_point_layer()
Determines whether the content of the TiledRasterRDD is of floating point type.

Parameters **None** –

Returns [boolean]

layer_metadata
Layer metadata associated with this layer.

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 A list of numpy arrays (the tiles)

Raises

- `ValueError` – If using lookup on a non `SPATIAL` `TiledRasterRDD`.
- `IndexError` – If col and row are not within the `TiledRasterRDD`'s bounds.

mask(*geometries*)

Masks the `TiledRasterRDD` so that only values that intersect the geometries will be available.

Parameters **geometries** (*list*) – A list of shapely geometries to use as masks.

> **Note:** All geometries must be in the same CRS as the TileLayer.

Returns *TiledRasterRDD*

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 that is contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon* or *shapely.geometry.MultiPolygon* or str) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKT string 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 that are contained within the given geometry.

Parameters **geometry** (*shapely.geometry.Polygon* or *shapely.geometry.MultiPolygon* or str) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKT string representation of the geometry.

Returns `float`

polygonal_min(*geometry*, *data_type*)

Finds the min value that is contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon* or *shapely.geometry.MultiPolygon* or str) – A Shapely `Polygon` or `MultiPolygon` that represents the area where the summary should be computed; or a WKT string 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 that are contained within the given geometry.

Parameters

- **geometry** (*shapely.geometry.Polygon* or *shapely.geometry.MultiPolygon* or str) – A Shapely Polygon or MultiPolygon that represents the area where the summary should be computed; or a WKT string 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(*start_zoom*, *end_zoom*, *resample_method='NearestNeighbor'*)
Creates a pyramid of GeoTrellis layers where each layer reprsents a given zoom.

Parameters

- **start_zoom** (*int*) – The zoom level where pyramiding should begin. Represents the level that is most zoomed in.
- **end_zoom** (*int*) – The zoom level where pyramiding should end. Represents the level that is most zoomed out.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants: NEARESTNEIGHBOR, BILINEAR, CUBICCONVOLUTION, LANCZOS, AVERAGE, MODE, MEDIAN, MAX, and MIN. If none is specified, then NEARESTNEIGHBOR is used.

Returns [TiledRasterRDDs].

Raises

- ValueError – If the given resample_method is not known.
- ValueError – If the col and row count is not a power of 2.

classmethod rasterize(*geopysc*, *rdd_type*, *geometry*, *extent*, *crs*, *cols*, *rows*, *fill_value*, *instant=None*)
Creates a TiledRasterRDD from a shapely geomety.

Parameters

- **geopysc** (GeoPyContext) – The GeoPyContext being used this session.
- **rdd_type** (*str*) – What the spatial type of the geotiffs are. This is represented by the constants: SPATIAL and SPACETIME.
- **geometry** (*str or shapely.geometry.Polygon*) – The value to be turned into a raster. Can either be a string or a Polygon. If the value is a string, it must be the WKT string, geometry format.
- **extent** (*Extent*) – The extent of the new raster.
- **crs** (*str or int*) – The CRS the new raster should be in.
- **cols** (*int*) – The number of cols the new raster should have.
- **rows** (*int*) – The number of rows the new raster should have.

- **fill_value** (*int*) – The value to fill the raster with.

  Note: Only the area the raster intersects with the `extent` will have this value. Any other area will be filled with GeoTrellis' NoData value for `int` which is represented in GeoPySpark as the constant, `NODATAINT`.

- **instant** (*int, optional*) – Optional if the data has no time component (ie is `SPATIAL`). Otherwise, it is requires and represents the time stamp of the data.

Returns *TiledRasterRDD*

Raises `TypeError` – If `geometry` is not a `str` or a Polygon; or if there was a mistach in inputs like setting the `rdd_type` as `SPATIAL` but also setting `instant`.

reclassify(*value_map*, *data_type*, *boundary_strategy='LessThanOrEqualTo'*, *replace_nodata_with=None*)

Changes the cell values of a raster based on how the data is broken up.

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`.
- **boundary_strategy** (*str, optional*) – How the cells should be classified along the breaks. This is represented by the following constants: `GREATERTHAN`, `GREATERTHANOREQUALTO`, `LESSTHAN`, `LESSTHANOREQUALTO`, and `EXACT`. If unspecified, then `LESSTHANOREQUALTO` will be used.
- **replace_nodata_with** (*data_type, 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: NoData symbolizes a different value depending on if `data_type` is `int` or `float`. For `int`, the constant `NODATAINT` can be used which represents the NoData value for `int` in GeoTrellis. For `float`, `float('nan')` is used to represent NoData.

Returns *TiledRasterRDD*

reproject(*target_crs*, *extent=None*, *layout=None*, *scheme='float'*, *tile_size=256*, *resolution_threshold=0.1*, *resample_method='NearestNeighbor'*)

Reproject RDD as tiled raster layer, samples surrounding tiles.

Parameters

- **target_crs** (*str or int*) – The CRS to reproject to. Can either be the EPSG code, well-known name, or a PROJ.4 projection string.
- **extent** (*Extent*, optional) – Specify the layout extent, must also specify `layout`.
- **layout** (*TileLayout*, optional) – Specify the tile layout, must also specify `extent`.

- **scheme** (*str, optional*) – Which LayoutScheme should be used. Represented by the constants: FLOAT and ZOOM. If not specified, then FLOAT is used.
- **tile_size** (*int, optional*) – Pixel dimensions of each tile, if not using layout.
- **resolution_threshold** (*double, optional*) – The percent difference between a cell size and a zoom level along with the resolution difference between the zoom level and the next one that is tolerated to snap to the lower-resolution zoom.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants: NEARESTNEIGHBOR, BILINEAR, CUBICCONVOLUTION, LANCZOS, AVERAGE, MODE, MEDIAN, MAX, and MIN. If none is specified, then NEARESTNEIGHBOR is used.

Note: extent and layout must both be defined if they are to be used.

Returns *TiledRasterRDD*

Raises TypeError – If either extent or layout is defined but the other is not.

stitch()
Stitch all of the rasters within the RDD into one raster.

Note: This can only be used on SPATIAL TiledRasterRDDs.

Returns *Raster*

tile_to_layout(*layout*, *resample_method='NearestNeighbor'*)
Cut tiles to a given layout and merge overlapping tiles. This will produce unique keys.

Parameters

- **layout** (*TileLayout*) – Specify the TileLayout to cut to.
- **resample_method** (*str, optional*) – The resample method to use for the reprojection. This is represented by the following constants: NEARESTNEIGHBOR, BILINEAR, CUBICCONVOLUTION, LANCZOS, AVERAGE, MODE, MEDIAN, MAX, and MIN. If none is specified, then NEARESTNEIGHBOR is used.

Returns *TiledRasterRDD*

to_numpy_rdd()
Converts a TiledRasterRDD 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 pyspark.RDD

unpersist()
Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

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.

zoom_level
The zoom level of the RDD. Can be `None`.

Python Module Index

g

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z