django-raster Documentation

Release 0.3

Daniel Wiesmann and Mike Flaxman

May 18, 2021

Contents

1 Contents 3

2 Indices and tables 17

Index 19

Django-raster provides raster data functionality for Django projects with a PostGIS database backend. It is based on the Django internal raster data type RasterField and GDAL bindings through GDALRaster.

The django-raster repository is hosted on GitHub.

CHAPTER 1

Contents

1.1 Installation

Django-raster requires `Django >= 1.9` configured with a PostGIS backend and the GDAL library. The use of Celery is highly recommended (see below).

The package is available on PyPI, you can install it with:

```
pip install django-raster
```

To integrate the package into Django, add `raster` to your `INSTALLED_APPS` setting like this:

```
INSTALLED_APPS = (
    ...
    'raster',
)
```

Django-raster has its own url structure (to serve raster data through a `TMS` endpoint for instance). To activate those urls, add the raster urls to your main urlconf:

```
urlpatterns = [
    ...
    url(r'^raster/', include('raster.urls')),
]
```

Finally, migrate your database to create the tables required by django-raster:

```
python manage.py migrate
```

1.1.1 Distributed Task Management

Django-raster works best with Celery, a distributed task queue manager. Parsing raster files is a process that will time out most of the time if done through regular http requests. If Celery is installed, several long running tasks will be

executed asynchronously in django-raster.

If you have Celery configured for your project, add the following to your project's settings to tell django-raster to use it:

```
RASTER_USE_CELERY = True
```

1.2 Introduction

Django-raster provides high level utilities to work with raster data in Django. It is based on the Django internal GDALRaster object and RasterField datatype.

There are three main components in this package:

- Parser utility to ingest rasters through the admin or the django shell.
- Tile Map Service (TMS) endpoint to render raster data.
- Raster calculator to compute and render raster calculator expressions.

Raster files are stored in a file field attached to `RasterLayer` objects. Data can be added by creating raster layers through the admin interface or the Django shell.

After creating a `RasterLayer` object, the raster data will be parsed automatically. The parsing can be executed asynchronously if Celery is integrated into the Django project. The raster parser will automatically extract the data in the raster and store it as PostGIS raster tiles on the database.

After ingesting the data, raster styles can be defined through the admin interface which are then used to render the data through TMS endpoints. The endpoints can be used in Javascript mapping software such as OpenLayers or Leaflet.

1.2.1 Limitations

The main limitation of the django-raster package is that it is focused on single band rasters. For most of the functionality, only the first band in the raster is used. While the tile parser processes and stores all bands of the input rasters, for the TMS endpoints and raster algebra calculations, currently only the first band is used.

Another limitation is that the projection of the raster tiles is fixed to the Web Mercator Projection (EPSG 3857). This is because a large part of online mapping applications use this projection, especially TMS services.

1.3 Raster Layers

A `RasterLayer` is django-raster's representation of raster files. It can be used to input raster data into your application. In most cases there is one `RasterLayer` for each raster file.

1.3.1 Storing a Raster File

Raster files can be uploaded through the admin interface and are stored in the `RasterLayer` model. Like for any other model, raster layers can also be created using the Django shell. Each raster file corresponds to one `RasterLayer` object. When adding a new raster file, the following properties are **required**:

- Layer name
- Raster file (either as file, http url or s3 url)
- Data type

The datatype tells django-raster how to interpret the pixel values. The choices are "continuous", "categorical", "mask", or "rank ordered". By default, django-raster extracts all other raster metadata from the input file. The **optional** input parameters are the following

- Description
- SRID
- Nodata value
- Max zoom value
- Legend

The srid, the nodata value and the maximum zoom value are all determined automatically from the raster properties if left blank. The max zoom value specifies the highest z-x-y zoom level to create tiles for (see below).

The legend attribute is a foreign key to a raster `Legend` object. If the raster legend is specified, it is used as default style when rendering tiles from that raster. How raster tiles are rendered is described in detail in the *Rendering tiles* section.

There are also three boolean flags that allow finer grained control over the raster layer parse process.

- Next higher zoom level
- Build pyramids
- Store reprojected

The raster layer is "snapped" to the next higher zoom level by default. To snap the raster to the next lower zoom level when compared to the true resolution of the data, the "next higher" flag has to be disactivated.

There is a "build pyramids" flag that controls whether the tiles should be created also for the lower zoom levels. This is enabled by default and is recommended in most cases as the tile renderer will expect those tiles to be present.

During parsing, the raster is reprojected to the web mercator projection. This operation is costly and is only done once by default. Django-raster stores a reprojected version in a separate model. To prevent the storage of the reprojected file, the "store reprojected" flag can be disactivated. Note that this will result in less use of storage, but an overhead when parsing, especially for asynchronous parsing where the file will be reprojected by each worker.

Specifying an Url as Source

The raster file can be uploaded directly using the raster file field, or passed as a url either to a public http(s) address, or a url like string, pointing directly to an s3 bucket. The http(s) urls are regular web urls.

For the s3 links, the `boto3` library is used to directly access an s3 bucket and download it from there. In this way, private or requester-pays buckets can be used as source. The credentials for accessing the buckets need to be configured so that boto3 can see them.

The url should have the following structure

```
s3://BUCKET_NAME/BUCKET_KEY
```

for instance,

```
s3://sentinel-s2-l1c/tiles/12/S/VG/2017/9/15/0/B12.jp2
```

gets the same file as the following regular http url

```
http://sentinel-s2-l1c.s3.amazonaws.com/tiles/12/S/VG/2017/9/15/0/B12.jp2
```

but instead of making a regular web request, it accesses the file using boto3.

Note that for requester pays bucket this might incur charges even if the requester is not the owner of the bucket.

1.3.2 Raster Tile Creation

Upon uploading a file, django-raster automatically parses the raster file. The parser extracts metadata from the raster and its bands, and creates tiles. The progress or possible errors in parsing is written to a parse log object, which is exposed on the `RasterLayer` admin interface.

The parser automatically creates a tile pyramid in the z-x-y scheme of a TMS. By default, the highest zoom level for which to create tiles is calculated automatically from the resolution of the raster. The zoom level is set such that the resolution of the highest zoom is at least the original resolution. This behavior can be changed by manually setting the highest zoom level, using the `max_zoom_value` field.

The tiles are stored as `RasterTile` objects. The raster data itself is stored as PostGIS rasters through a RasterField. The tiles are managed automatically through their parent `RasterLayer` object, and do normally not require any manual user manipulation.

Asynchronous Parsing

It is highly recommended to configure the Django application with Celery, to parse the rasters asynchronously. For most raster files, the creation of tiles takes several minutes or even hours to complete. Since the parsing is triggered automatically upon upload, the html requests in the admin will often time out. For more information about how to configure Celery, consult the *Installation* section.

1.4 Rendering tiles

After creating and parsing a `RasterLayer`, the tiles for that layer can be accessed through the tiles url. The raster urls have to be added to the application's url patterns. Here we assume that the `/raster/` base url is used as proposed in the *Installation* section.

The tiles url is structured as follows,

```
/raster/tiles/layer_id/{z}/{x}/{y}.png
```

where the `layer_id` is the primary key of a raster layer. This structure can be used directly in online mapping software such as OpenLayers or Leaflet. An example request could look like this: `/raster/tiles/23/8/536/143.png`, returning a tile in png format of the layer with ID `pk=23` at zoom level `z=8` and indexes `x=536` and `y=143`.

By default, the tiles are rendered using simple grayscale. To apply a custom colormap, a `Legend` needs to be assigned to the layer. Raster layers have an optional foreign key to a Legend object, which can be set through the admin interface.

1.4.1 Legends

Legends are objects that are used to interpret raster data. This includes the cartographic information (colors), but also the semantics of the data (such as names). Legends be created through the admin interface.

A legend is stored as in the `Legend` model, which is a collection of `LegendEntry` objects. Each of the entries have an expression for classifying the data and a semantic meaning of the expression. The semantics of the expression are stored in the `LegendSemantics` model. Here is an example for a legend representing two temperatures:

```
>>> from raster.models import Legend, LegendEntry, LegendEntryOrder, LegendSemantics
>>> hot_semantics = LegendSemantics.objects.create(name='Hot')
>>> cold_semantics = LegendSemantics.objects.create(name='Cold')
>>> hot_entry = LegendEntry.objects.create(semantics=cold, expression='0', color='
↪#0000FF')
>>> cold_entry = LegendEntry.objects.create(semantics=hot, expression='1', color='
↪#FF0000')
>>> legend = Legend.objects.create(title='Temperatures')
>>> LegendEntryOrder.objects.create(legend=legend, legendentry=entry, code='1')
>>> legend.json
... '[{"color": "#FFFFFF", "expression": "1", "name": "Earth"}]'
```

Legend Entries

`LegendEntry` entries relate semantics and a color value with a range of pixel values. One entry has a foreign key to a `LegendSemantics` object, a color in hex format and an expression.

The expression is a classification of pixels. It describes a range of pixel values in the data. It is either an exact number for discrete rasters, or a formula for continuous rasters:

```
expression = "3"  # Matches all pixels with an exact value of 3
```

For more complicated expressions, a logical expression can be specified through a formula. The variable `x` represents the pixel value in the formula. Here are two examples of valid formula expressions:

```
# Match pixel values bigger than -3 and smaller or equal than 1
expression = "(-3.0 < x) & (x <= 1)"
# Match all pixels with values smaller or equal to one
expression = "x <= 1"
```

Formula expressions are currently not validated on input. Wrongly specified formulas might lead to errors when rendering raster tiles. Check your formulas if unexpected errors happen on the TMS endpoints.

Continuous Color Schemes

The examples above show how to assign discrete pixel value ranges to individual colors. This allows applying discrete color schemes with a limited number of breaks to continuous rasters.

Django-raster also supports applying continuous color scales. Colormaps are interpreted as continuous color schemes if the keyword `continuous` is provided as a key in the colormap dictionary.

The continuous color scheme requires at least two colors, which are interpolated over the range of pixel values. These colors can be specified using the `from` and `to` keywords. A third color can be specified to force interpolation through another color in the middle of the range. This intermediate color can be specified using the `over` key.

The range over which the colors are interpolated is determined automatically from the raster layer metadata if possible, and falls back to the range of the individual tile data. The fallback might result in a visually confusing color scheme, as the range of pixel values in a single tile may vary substantially and are not representative of the raster. The range can therefore also be specified manually using the `range` parameter.

An example for a continuous color scheme, which will interpolate all values from `0` to `100` into colors ranging from red to blue over green is the following:

```
{
    "continuous": "True",
```

(continues on next page)

(continued from previous page)

```
    "from": [255, 0, 0],
    "to": [0, 0, 255],
    "over": [0, 255, 0],
    "range": [0, 100]
}
```

The keys `continuous`, `from` and `to` are required. The `over` key is an optional intermediate color for the interpolation. The `range` key specifies the pixel values over which to interpolate. This parameter is estimated from metadata if not provided in the legend. All other keys are ignored in the continuous color mode, which is triggered if the `continuous` key is found in the legend.

1.4.2 Overriding the colormap and the legend

While a legend and a colormap can be associated with a raster layer objects in the database it is nonetheless possible to overwrite the legend or colormap used to render the tiling. Overriding is done via the following url parameters:

Parameter	Description
legend	Use given legend to render the tiles
store	One of `database`, `session`. Fetch legend from database or session, default is database
colormap	Overrides the raster layer's legend colormap.

Examples

If you want to overrides the legend to use MyOtherLegend stored in database you can use the following url for the tiling (assuming *23* is your rasterlayer_id):

```
/raster/tiles/23/{z}/{x}/{y}.png?legend=MyOtherLegend
```

If you want to use the legend from the session with the same name as above you can use following one:

```
/raster/tiles/23/{z}/{x}/{y}.png?legend=MyOtherLegend&store=session
```

Note: You can set and get a session colormap with the help of shortcuts functions *set_session_colormap()* and *get_session_colormap()*.

And finally if you want to provide this custom colormap

```
{
    "1": "#FF0000",
    "2": "#00FF00",
    "3": "#0000FF"
}
```

you can do so by using this url:

```
/raster/tiles/{z}/{x}/{y}.png?colormap=%22%7B1%3A%20'%23FF0000'%2C%202%3A%20'%2300FF00
↪'%2C%203%3A%20'%230000FF'%7D%22
```

The colormap value is the URIEncoded version of the json stringified colormap object.

1.4.3 Image formats

All endpoints (regular tiles, algebra and RGB) support three formats: PNG, JPEG and TIFF. The different formats can be requested by changing the file extension in the url. The extensions to use are `.png`, `.jpg`, and `.tif`.

The PNG and JPEG endpoints behave the same way, except that JPEG images do not support an alpha channel. Nodata pixels are rendered in black.

The TIFF endpoint will return the raw data produced from the request in a georeferenced GeoTIFF file. It therefore ignores any of the rendering parameters and simply returns the raw values of the result of the request. This might be useful for analysis purposes, where raster algebra results can be obtained in raw form for further downstream processing.

1.5 Raster Algebra

Django-raster has raster calculator functionality. The raster calculator allows rendering raster tiles based on algebraic formulas. The use is very similar to a standard z/x/y tile endpoint, but allows the evaluation of a broad range of algebraic expressions applied to existing pixel values. The z/x/y structure can be used directly in online mapping software such as OpenLayers or Leaflet.

Similar to the regular tiles endpoint, the django-raster url patterns need to be installed for the raster algebra endpoint to work. For the documentation we assume that the `/raster/` base url is used as proposed in the *Installation* section.

1.5.1 Raster algebra TMS endpoint

The raster algebra url base is used only to specify the z/x/y tile index. All the rest of the configuration is done through the query parameters. The input to the raster algebra is a named list of `RasterLayer` ids and a formula for evaluation. These values are passed to the backend through two required query parameters: `layers` and `formula`.

The layers query parameter identifies which raster layers to use for evaluation. It is a comma separated list of variable-name and RasterLayer id pairs. The variable names are matched with the names in the formula. An example is `layers=a=2,b=4` which will match `RasterLayer` with id `2` to variable name `a` and the layer with id `4` with the variable name `b`.

The formula query parameter is a string specifying a formula for evaluation. The formula is an algebraic expression based on the names given to the layers in the `layers` query parameter. The formula has to be an expression that can be evaluated by the `FormulaParser`. It accepts a broad range of algebraic expressions. The endpoint supports most of the common mathematical operators (`+`, `-`, `*`, `/`, etc), functions (`sin`, `cos`, `exp`, etc.), and logical operators (`&`, `!`, `>`, `=`, etc.). It also has a set of predefined constants through reserved keywords such as pi `PI` or the Euler number `E`.

Putting it all together, an example request to the raster algebra endpoint could look like this:

```
/raster/algebra/{z}/{x}/{y}.png?layers=a=1,b=3,c=6&formula=log(a+b)*c&legend=5
```

In addition to the required query parameters: `layers` and `formula`, a `Legend` id can be specified using the `legend` query parameter. If specified, the legend will be used to interpret the result of the algebra expression. This is convenient to use predefined colormaps for the endpoint.

Dynamic colormap

For a more dynamic rendering scheme, a dynamic colormap can be passed to the endpoint using the `colormap` query parameter. The following request would color all pixels that result in a value bigger than zero in red, and all other pixels in green.

```
/raster/algebra/{z}/{x}/{y}.png?layers=a=1,b=3,c=6&formula=log(a+b)*c&colormap={'x>0':
↪'#FF0000','x<=0':'#00FF00'}
```

Using specific bands

By default, the algebra and rgb endpoints use the first band in each layer specified. To use a specific band, use a `'variable:band'` syntax, where variable is the name of the variable, and band is the band index. For example `{'a:3': 23}` would match band 3 of the `RasterLayer` with the id `23` to the variable name `a`.

Encoding

Both the colormap and the formula should be properly url encoded. The examples here are not encoded and should be considered as instructive examples only.

RGB endpoint

The algebra endpoint can also be used to render RGB images. For this, only three query parameters are expected: `r`, `g`, and `b`. If these three parameters are found in the list of query parameters, and no formula has been specified, the three input bands are interpreted as RGB channels of an RGB image. For example to use raster layer with id `1` as red, id `3` as green and id `6` as blue, the following url can be used:

```
/raster/algebra/{z}/{x}/{y}.png?layers=r=1,g=3,b=6
```

If the raw data in the tiles is not already scaled to the range [0, 255], an additional scaling factor can be specified, which will be used to rescale all three bands to the default RGB color range. For instance, the following query would assume that the input bands have values in the range of [5, 10000], and would rescale them to the RGB color space.

```
/raster/algebra/{z}/{x}/{y}.png?layers=r=1,g=3,b=6&scale=5,10000
```

An alpha channel can be activated by passing the `alpha` query parameter. The alpha parameter makes all the pixels transparent that have values equal to `0` in all three RGB channels.

For multi band rasters that have the rgb channels as bands and not in separate files, the band accessor syntax can be used. For instance, if the layer with id `23` is a 3-band RGB raster, the following would render the tiles as RGB using bands `0`, `1`, and `2`:

```
/raster/algebra/{z}/{x}/{y}.png?layers=r:0=23,g:1=23,b:2=23
```

Image Enhancement

The algebra and TMS endpoints support image enhancement using the `ImageEnhance` PIL module. The following query parameters arguments are passed to the corresponding image enhancers. The parameter value is passed to the enhancer as `factor` argument.

Table 1: Enhancer query parameters.

Query	Enhancer
enhance_color	ImageEnhance.Color
enhance_contrast	ImageEnhance.Contrast
enhance_brightness	ImageEnhance.Brightness
enhance_sharpness	ImageEnhance.Sharpness

The following example enhances the contrast of tiles from the RGB endpoint by a factor of 3:

```
/raster/algebra/{z}/{x}/{y}.png?layers=r=1,g=3,b=6&scale=5,10000&enhance_contrast=3
```

1.5.2 Pixel Value Lookup

Single pixel values for raster algebra expressions can be looked up by coordinates. The endpoint works very similar to the raster algebra TMS endpoint, but instead of Z-X-Y tile indices, coordinates are passed through the url. The query parameters are analogue to the algebra TMS endpoint as described above.

The base url structure is

```
/raster/pixel/{xcoord}/{ycoord}/
```

For instance, the following request will return the pixel value of the requested raster algebra expressino for the coordinates `xcoord = -9218229` and `ycoord = 3229269`. The coordinates must be provided in the web mercator projection (EPSG 3857).

```
/raster/pixel/-9218229/3229269/?layers=a=1,b=3,c=6&formula=log(a+b)*c
```

1.5.3 Formula parser

At the heart of the raster calculator is the `FormulaParser`, which is based on the pyparsing package. The `FormulaParser` is a general purpose formula evaluation class. It is It does not know about rasters and operates with Numpy arrays directly. To use it, you need a dictionary with Numpy arrays of equal shape and a formula as string. The keys in the dictionary are the variable names and are used to match data to variables in the formula. Here are some examples of how to use the formula parser:

```
# Import parser and instantiate an instance.
>>> from raster.algebra.parser import FormulaParser
>>> parser = FormulaParser()
# Create a data dictionary and evaluate a simple sum.
>>> data = {'a': range(5), 'b': range(5)}
>>> formula = 'a + b'
>>> parser.evaluate(data, formula)
... array([0, 2, 4, 6, 8])
# Use the sin function and divide by b.
>>> formula = 'sin(a) / b'
>>> parser.evaluate(data, formula)
... array([ nan, 0.84147098, 0.45464871, 0.04704, -0.18920062])
# Use a logical array.
>>> data.update({'a_new_var': [True, False, False, True, False]})
>>> formula = '!a_new_var * a + 3'
>>> parser.evaluate(data, formula)
... array([ 3.,  4.,  5.,  3.,  7.])
# Use the PI keyword in a formula.
>>> formula = 'a * PI'
>>> parser.evaluate(data, formula)
>>> array([0. , 3.14159265, 6.28318531, 9.42477796, 12.56637061])
```

1.5.4 Raster algebra parser

The `RasterAlgebraParser` class is a wrapper that can be used to apply the generic formula parser to raster objects directly. The use is identical to the generic case except that the objects in the data dictionary are expected to

be :class:GDALRaster objects. The data arrays are extracted from the raster objects automatically and are passed to the formula parser. The result array is converted into a `GDALRaster` before returning.

By default, the first band is used for calculations, to specify a specific band to be used the syntax is `'variable:band'`, where variable is the name of the variable, and band is the band index. For example `{'a:3': rst}` would match band 3 of the GDALRaster `rst` to the variable name `a`.

Here is a complete example for how to use the `RasterAlgebraParser`.

```
>>> from raster.algebra.parser import RasterAlgebraParser
>>> parser = RasterAlgebraParser()
>>> base = {
>>>     'datatype': 1,
>>>     'driver': 'MEM',
>>>     'width': 2,
>>>     'height': 2,
>>>     'srid': 3086,
>>>     'origin': (500000, 400000),
>>>     'scale': (100, -100),
>>>     'skew': (0, 0),
>>>     'bands': [
>>>         {'nodata_value': 10},
>>>         {'nodata_value': 10},
>>>         {'nodata_value': 10},
>>>     ],
>>> }
>>> base['bands'][0]['data'] = range(20, 24)
>>> base['bands'][1]['data'] = range(10, 14)
>>> rast1 = GDALRaster(base)
>>> base['bands'][0]['data'] = [1, 1, 1, 1]
>>> rast2 = GDALRaster(base)
>>> base['bands'][0]['data'] = [30, 31, 32, 33]
>>> base['bands'][0]['nodata_value'] = 31
>>> rast3 = GDALRaster(base)
>>> data = dict(zip(['x:1', 'y:0', 'z'], [rast1, rast2, rast3]))
>>> rst = parser.evaluate_raster_algebra('x*(x>11) + 2*y + 3*z*(z==30)')
>>> rst.bands[0].data()
... array([[ 10.,  10.],
...        [ 14.,  15.]])
```

1.5.5 Keywords, Operators and Functions

The following tables list the available operators, functions and reserved keywords from the `FormulaParser` and the corresponding raster calculator.

Table 2: Keyword symbols

Keyword	Symbol
Euler Number	`E`
Pi	`PI`
True Boolean	`TRUE`
False Boolean	`FALSE`
Null	`NULL`
Infinite	`INF`

Table 3: Operator symbols

Operator	Symbol
Add	`+`
Substract	`-`
Multiply	`*`
Divide	`/`
Power	`^`
Equal	`==`
Not Equal	`!=`
Greater	`>`
Greater or Equal	`>=`
Less	`<`
Less or Equal	`<=`
Logical Or	`|`
Logial And	`&`
Logcal Not	`!`
Fill Nodata Values	`~`
Unary And	`+`
Unary Minus	`-`
Unary Not	`!`

Table 4: Function symbols

Function	Symbol
Sinus	`sin`
Cosinus	`cos`
Tangens	`tan`
Natural Logarithm	`log`
Exponential Function	`exp`
Absolute Value	`abs`
Integer	`int`
Round	`round`
Sign	`sign`
Minimum	`min`
Maximum	`max`
Mean	`mean`
Median	`median`
Standard Deviation	`std`
Sum	`sum`

1.6 Settings

A list of available settings to customize django-raster's behavior.

1.6.1 Asynchronous raster parsing

Determines whether to use celery tasks for parsing raster layers. It is highly recommended to configure celery, as raster parsing can take quite a while and the parsing through normal web requests will often timed out, even for medium sized raster.

```
RASTER_USE_CELERY = False
```

1.6.2 Parser working directory

Use this to specify a custom working directory used by the django-raster package when parsing raster files. This is where intermediate files are stored. Defaults to the normal temporary directory of the machine.

```
RASTER_WORKDIR = None
```

1.6.3 All in one parse task

For some applications where the size of the rasters is small, the distributed raster parsing might have more overhead than gain. The distributed parsing can be disactivated with the following setting. If it is set to `True`, rasters are parsed in single celery tasks.

```
RASTER_PARSE_SINGLE_TASK = True
```

1.6.4 Parse Batch Size

During parsing of a raster, tiles are written to the database in batches, using the `bulk_create` method. The size of each batch in the loop can be controlled by using the setting below. Defaults to 500 tiles.

```
RASTER_BATCH_STEP_SIZE = 500
```

1.6.5 S3 Endpoint URL

Set the S3 compatible endpoint used when retrieving raster tile sources from S3.

```
RASTER_S3_ENDPOINT_URL = "http://localhost:4572"
```

1.7 API Reference

1.7.1 Shortcuts

set_session_colormap(*session*, *key*, *colormap*)
Store the colormap in the user session.

get_session_colormap(*session*, *key*)
Get the colormap form a legend stored in the user session and identified by key.

1.7.2 Raster Utilities

Django-raster hosts some utilities that ease the interaction with raster data. The functions are located in `raster.utils` and `raster.tiles.utils`.

pixel_value_from_point (*raster*, *point*, *band=0*)

Return the pixel value for the coordinate of the input point from selected band.

The input can be a point or tuple, if its a tuple it is assumed to be a pair of coordinates in the reference system of the raster. The band index to be used for extraction can be specified with the `band` keyword.

Example:

```
# Create a raster.
>>> raster = GDALRaster({
    'width': 5,
    'height': 5,
    'srid': 4326,
    'bands': [{'data': range(25)}],
    'origin': (2, 2),
    'scale': (1, 1)
})
# Create a point at origin
>>> point = OGRGeometry('SRID=4326;POINT(2 2)')
# Get pixel value at origin.
>>> pixel_value_from_point(raster, point)
... 0
# Get pixel value from within the raster, using coordinate tuple input.
>>> pixel_value_from_point(raster, (2, 3.5))
... 5
```

CHAPTER 2

Indices and tables

- genindex
- modindex

Index

G

`get_session_colormap()` (*built-in function*), 14

P

`pixel_value_from_point()` (*built-in function*), 14

S

`set_session_colormap()` (*built-in function*), 14