# GeoPandas

GeoPandas: Pandas + geometry data type + custom geo

## Background

GeoPandas adds a spatial geometry data type to Pandas and enables spatial operations on these types, using shapely. GeoPandas leverages Pandas together with several core open source geospatial packages and practices to provide a uniquely simple and convenient framework for handling geospatial feature data, operating on both geometries and attributes jointly, and as with Pandas, largely eliminating the need to iterate over features (rows). Also, as with Pandas, it adds a very convenient and fine-tuned plotting method, and read/write methods that handle multiple file and “serialization” formats.

GeoPandas builds on mature, stable and widely used packages (Pandas, shapely, etc). It is being supported more and more as a preferred Python data structure for geospatial vector data.

**When should you use GeoPandas?**

* For exploratory data analysis, including in Jupyter notebooks.
* For highly compact and readable code. Which in turn improves reproducibility.
* If you’re comfortable with Pandas, R dataframes, or tabular/relational approaches.

**When it may not be the best tool?**

* For polished map creation and multi-layer, interactive visualization; if you’re comfortable with GIS software, one option is to use a desktop GIS like QGIS! You can generate intermediate GIS files and plots with GeoPandas, then shift over to QGIS. Or refine the plots in Python with matplotlib or additional packages.
* If you need very high performance. Performance has been increasing and substantial enhancements are in the works (including possibly a Dask parallelization implementation).

## Set up packages and data file path

%matplotlib inline

**from** **\_\_future\_\_** **import** (absolute\_import, division, print\_function)

**import** **os**

**import** **matplotlib** **as** mpl

**import** **matplotlib.pyplot** **as** plt

**from** **shapely.geometry** **import** Point

**import** **pandas** **as** pd

**import** **geopandas** **as** gpd

**from** **geopandas** **import** GeoSeries, GeoDataFrame

data\_pth = "../data"

mpl.\_\_version\_\_, pd.\_\_version\_\_, gpd.\_\_version\_\_

('2.2.3', '0.23.4', '0.4.0')

## GeoSeries: The geometry building block

Like a Pandas Series, a GeoSeries is the building block for the more broadly useful and powerful GeoDataFrame that we’ll focus on in this tutorial. Here we’ll first take a bit of time to examine a GeoSeries.

A GeoSeries is made up of an index and a GeoPandas geometry data type. This data type is a shapely.geometry object, and therefore inherits their attributes and methods such as area, bounds, distance, etc.

GeoPandas has six classes of geometric objects, corresponding to the three basic single-entity geometric types and their associated homogeneous collections of multiple entities:

* + Single entity (core, basic types):
    - Point
    - Line (formally known as a LineString)
    - Polygon
  + Homogeneous entity collections:
    - Multi-Point
    - Multi-Line (MultiLineString)
    - Multi-Polygon

A GeoSeries is then a list of geometry objects and their associated index values.

**Entries (rows) in a GeoSeries can store different geometry types**

GeoPandas does not constrain the geometry column to be of the same geometry type. This can lead to unexpected problems if you’re not careful! Specially if you’re used to thinking of a GIS file format like shape files, which store a single geometry type. Also beware that certain export operations (say, to shape files …) will fail if the list of geometry objects is heterogeneous.

### ****Create a GeoSeries from a list of shapely Point objects constructed directly from WKT text****

**from** **shapely.wkt** **import** loads

GeoSeries([loads('POINT(1 2)'), loads('POINT(1.5 2.5)'), loads('POINT(2 3)')])

0 POINT (1 2)

1 POINT (1.5 2.5)

2 POINT (2 3)

dtype: object

### ****Create a GeoSeries from a list of shapely Point objects using the Point constructor****

gs = GeoSeries([Point(-120, 45), Point(-121.2, 46), Point(-122.9, 47.5)])

gs

0 POINT (-120 45)

1 POINT (-121.2 46)

2 POINT (-122.9 47.5)

dtype: object

type(gs), len(gs)

(geopandas.geoseries.GeoSeries, 3)

A GeoSeries (and a GeoDataFrame) can store a CRS implicitly associated with the geometry column. This is useful as essential spatial metadata and for transformation (reprojection) to another CRS.

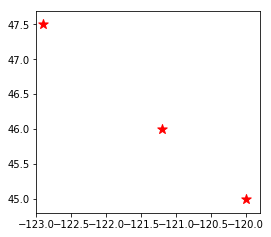
gs.crs = {'init': 'epsg:4326'}

The plot method accepts standard matplotlib.pyplot style options, and can be tweaked like any other matplotlib figure.

gs.plot(marker='\*', color='red', markersize=100, figsize=(4, 4))

plt.xlim([-123, -119.8])

plt.ylim([44.8, 47.7]);



We’ll define a simple dictionary of lists, that we’ll use again later.

data = {'name': ['a', 'b', 'c'],

'lat': [45, 46, 47.5],

'lon': [-120, -121.2, -122.9]}

Note this convenient, compact approach to create a list of Point shapely objects out of X & Y coordinate lists (an alternate approach is shown in the Advanced notebook):

geometry = [Point(xy) **for** xy **in** zip(data['lon'], data['lat'])]

geometry

[<shapely.geometry.point.Point at 0x7fb9438989e8>,

<shapely.geometry.point.Point at 0x7fb9438b46d8>,

<shapely.geometry.point.Point at 0x7fb9438b4e10>]

We’ll wrap up by creating a GeoSeries where we explicitly define the index values.

gs = GeoSeries(geometry, index=data['name'])

gs

a POINT (-120 45)

b POINT (-121.2 46)

c POINT (-122.9 47.5)

dtype: object

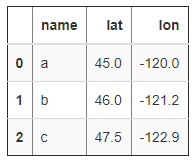
## GeoDataFrames: The real power tool

### Start with a simple, manually constructed illustration

We’ll build on the GeoSeries examples. Let’s reuse the data dictionary we defined earlier, this time to create a DataFrame.

df = pd.DataFrame(data)

df



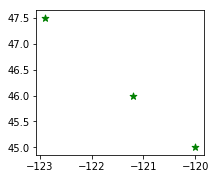
Now we use the DataFrame and the “list-of-shapely-Point-objects” approach to create a GeoDataFrame. Note the use of two DataFrame attribute columns, which are effectively just two simple Pandas Series.

geometry = [Point(xy) **for** xy **in** zip(df['lon'], df['lat'])]

gdf = GeoDataFrame(df, geometry=geometry)

There’s nothing new to visualize, but this time we’re using the plot method from a GeoDataFrame, not from a GeoSeries. They’re not exactly the same thing under the hood.

gdf.plot(marker='\*', color='green', markersize=50, figsize=(3, 3));

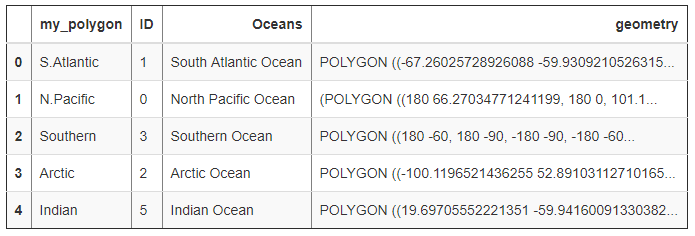


### Load and examine the simple “oceans” shape file

gpd.read\_file is the workhorse for reading GIS files. It leverages the fiona package.

oceans = gpd.read\_file(os.path.join(data\_pth, "oceans.shp"))

oceans.head()



The crs was read from the shape file’s prj file:

oceans.crs {"init": "epsg:4326"}

Now we plot a real map, from a dataset that’s global-scale and stored in “geographic” (latitude & longitude) coordinates. It’s not the actual ocean shapes defined by coastal boundaries, but bear with me. A colormap has been applied to distinguish the different Oceans.

oceans.plot(cmap='Set2', figsize=(10, 10));

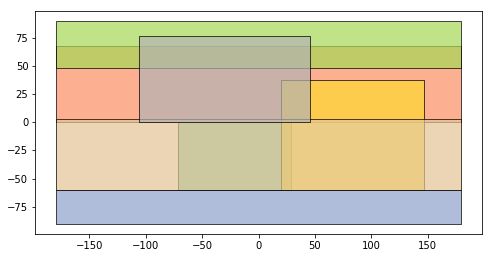


**oceans.shp** stores both Polygon and Multi-Polygon geometry types (a Polygon may also be viewed as a Multi-Polygon with 1 member). We can get at the geometry types and other geometry properties easily.



The envelope method returns the bounding box for each polygon.

oceans.envelope.plot(cmap='Set2', figsize=(8, 8), alpha=0.7, edgecolor='black');



Does it seem weird that some envelope bounding boxes, such as the North Pacific Ocean, span all longitudes? That’s because they’re Multi-Polygons with edges at the ends of the -180 and +180 degree coordinate range.

oceans[oceans['Oceans'] == 'North Pacific Ocean'].plot(figsize=(8, 8));

plt.ylim([-10, 100]);

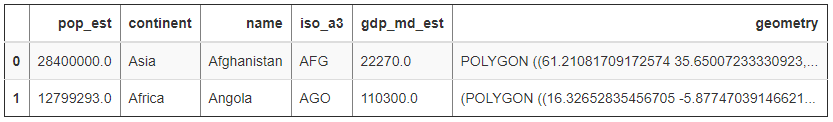


### Load “Natural Earth” countries dataset, bundled with GeoPandas

“*Natural Earth* is a public domain map dataset available at 1:10m, 1:50m, and 1:110 million scales. Featuring tightly integrated vector and raster data.” A subset comes bundled with GeoPandas and is accessible from the gpd.datasets module. We’ll use it as a helpful global base layer map.

world = gpd.read\_file(gpd.datasets.get\_path('naturalearth\_lowres'))

world.head(2)

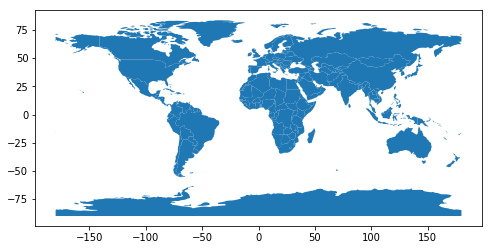


Its CRS is also EPSG:4326:

world.crs

{"init": "epsg:4326"}

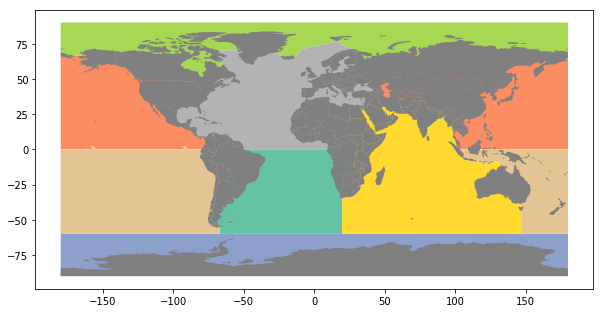
world.plot(figsize=(8, 8));



### Map plot overlays: Plotting multiple spatial layers

Here’s a compact, quick way of using the GeoDataFrame plot method to overlay two GeoDataFrames while customizing the styles for each layer.

world.plot(ax=oceans.plot(cmap='Set2', figsize=(10, 10)), facecolor='gray');



We can also compose the plot using conventional matplotlib steps and options that give us more control.

f, ax = plt.subplots(1, figsize=(12, 6))

ax.set\_title('Countries and Ocean Basins')

*# Other nice categorical color maps (cmap) include 'Set2' and 'Set3'*

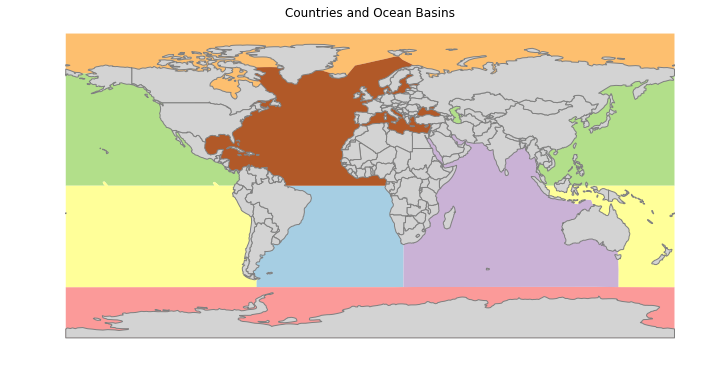
oceans.plot(ax=ax, cmap='Paired')

world.plot(ax=ax, facecolor='lightgray', edgecolor='gray')

ax.set\_ylim([-90, 90])

ax.set\_axis\_off()

plt.axis('equal');



## Extras: Reading from other data source types; fancier plotting

### Read PostgreSQL/PostGIS dataset from the Amazon Cloud

The fact that it’s on an Amazon Cloud is irrelevant. The approach is independent of the location of the database server; it could be on your computer.

**import** **json**

**import** **psycopg2**

First we’ll read the database connection information from a hidden JSON file, to add a level of security and not expose all that information on the github GeoHackWeek repository. This is also a good practice for handling sensitive information.

**with** open(os.path.join(data\_pth, "db.json")) **as** f:

db\_conn\_dict = json.load(f)

Open the database connection, returning a connection object:

conn = psycopg2.connect(\*\*db\_conn\_dict)

Now that we’ve used the connection information, we’ll overwrite the user and password keys (for security) and print out the dictionary, to give you a look at what needs to be in it:

db\_conn\_dict['user'] = '\*\*\*\*\*'

db\_conn\_dict['password'] = '\*\*\*\*\*'

db\_conn\_dict

{"host": "dssg2017.csya4zsfb6y4.us-east-1.rds.amazonaws.com",

"port": 5432,

"database": "geohack",

"user": "\*\*\*\*\*",

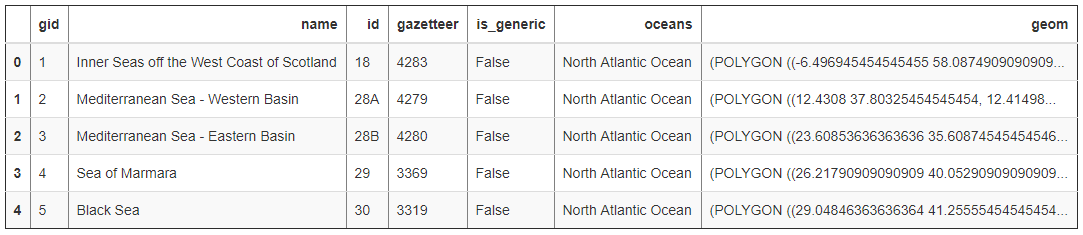
"password": "\*\*\*\*\*"}

Read in the world\_seas PostGIS dataset (a spatially enabled table in the PostgreSQL database) into a GeoDataFrame, using the opened connection object.

seas = gpd.read\_postgis("select \* from world\_seas", conn,coerce\_float=False)

seas.crs {"init": "epsg:4326"}

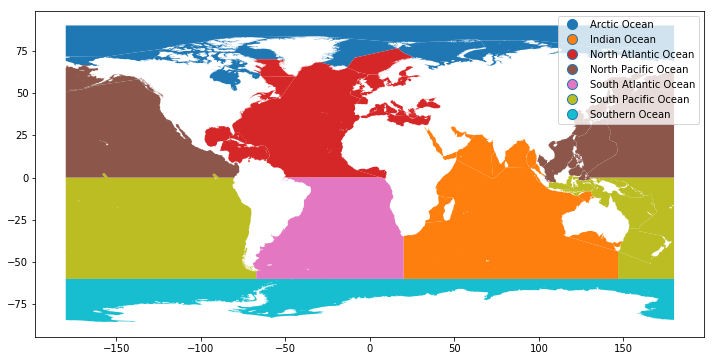
Let’s take a look at the GeoDataFrame.



### More advanced plotting and data filtering

Color the layer based on one column that aggregates individual polygons; using a categorical map, as before, but explicitly selecting the column and categorical mapping displaying an auto-generated legend, while displaying all polygon boundaries. Each “oceans” entry contain one or more ‘seas’.

seas.plot(column='oceans', categorical=True, legend=True, figsize=(14, 6));



## Additional plotting examples

Combine what we’ve learned. A map overlay, using world as a background layer, and filtering seas based on an attribute value (from oceans column) and an auto-derived GeoPandas geometry attribute (area). **world is in gray, while the filtered seas is in color.**

seas\_na\_arealt1000 = seas[(seas['oceans'] == 'North Atlantic Ocean')

& (seas.geometry.area < 1000)]

seas\_na\_arealt1000.plot(ax=world.plot(facecolor='lightgray', figsize=(8, 8)),

cmap='Paired', edgecolor='black')

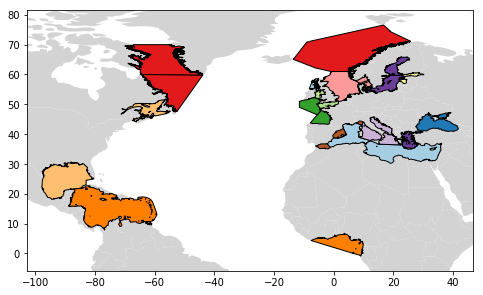
*# Use the bounds geometry attribute to set a nice*

*# geographical extent for the plot, based on the filtered GeoDataFrame*

bounds = seas\_na\_arealt1000.geometry.bounds

plt.xlim([bounds.minx.min()-5, bounds.maxx.max()+5])

plt.ylim([bounds.miny.min()-5, bounds.maxy.max()+5]);



### Save the filtered seas GeoDataFrame to a shape file

The to\_file method uses the fiona package to write to a GIS file. The default driver for output file format is ‘ESRI Shapefile’, but many others are available because fiona leverages GDAL/OGR.

seas\_na\_arealt1000.to\_file(os.path.join(data\_pth, "seas\_na\_arealt1000.shp"))

### Read from OGC WFS GeoJSON response into a GeoDataFrame

Use an Open Geospatial Consortium (OGC) Web Feature Service (WFS) request to obtain geospatial data from a remote source. OGC WFS is an open geospatial standard.

We won’t go into all details about what’s going on. Suffice it to say that we issue an OGC WFS request for all features from the layer named “oa:goainv” found in a GeoServer instance from NANOOS, requesting the response in GeoJSON format. Then we use the geojson package to “load” the raw response (a GeoJSON string) into a geojson feature object (a dictionary-like object).

**import** **requests**

**import** **geojson**

wfs\_url = "http://data.nanoos.org/geoserver/ows"

params = dict(service='WFS', version='1.0.0', request='GetFeature',

typeName='oa:goaoninv', outputFormat='json')

r = requests.get(wfs\_url, params=params)

wfs\_geo = geojson.loads(r.content)

Let’s examine the general characteristics of this GeoJSON object

**print**(type(wfs\_geo))

**print**(wfs\_geo.keys())

**print**(len(wfs\_geo.\_\_geo\_interface\_\_['features']))

<class 'geojson.feature.FeatureCollection'>

dict\_keys(['type', 'totalFeatures', 'crs', 'features'])

596

Now use the from\_features constructor method to create a GeoDataFrame directly from the geojson.feature.FeatureCollection object.

wfs\_gdf = GeoDataFrame.from\_features(wfs\_geo)

Visualize the data set as a simple map overlay plot and display the values for the last feature.

wfs\_gdf.plot(ax=world.plot(cmap='Set3', figsize=(10, 6)),

marker='o', color='red', markersize=15);

