GeoPandas

GeoPandas: Pandas + geometry data type + custom geo

---

1. 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).

---

2. 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')
```

3. 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

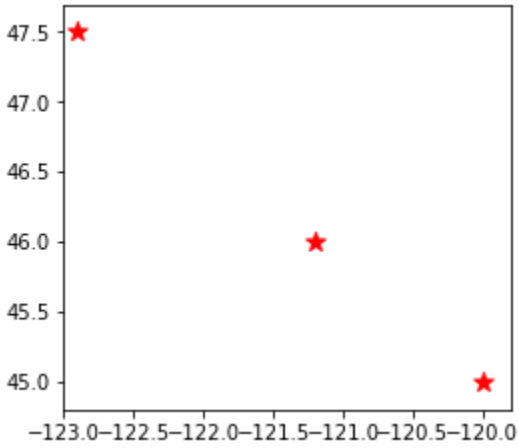
```
gs = GeoSeries([Point(-120, 45), Point(-121.2, 46), Point(-122.9, 47.5)])
gs
0        POINT (-120 45)
1      POINT (-121.2 46)
2    POINT (-122.9 47.5)
dtype: object
type(gs), len(gs)
(geopandas.geoseries.GeoSeries, 3)
```

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
```

5

4. 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
```

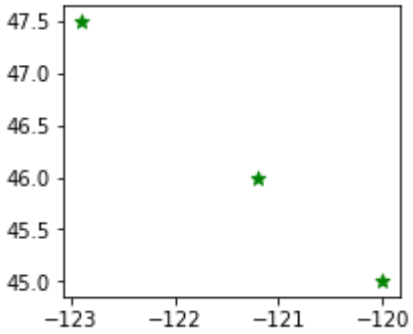
	name	lat	lon
0	a	45.0	-120.0
1	b	46.0	-121.2
2	c	47.5	-122.9

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()
```

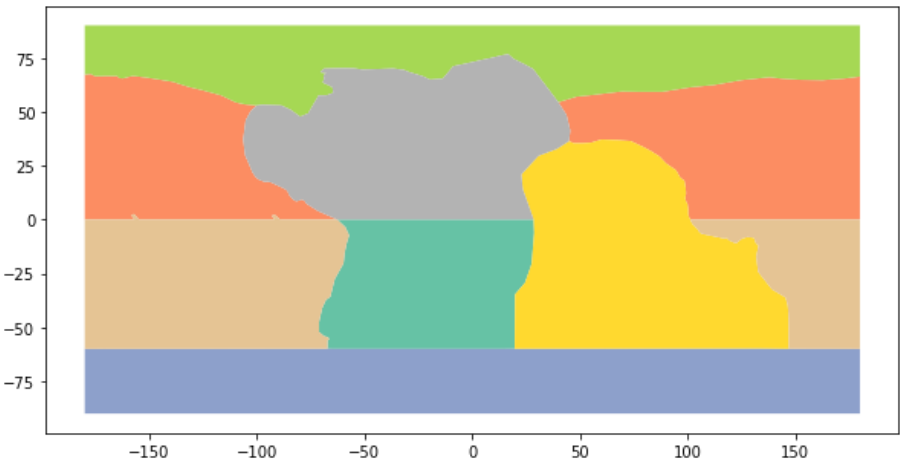
	my_polygon	ID	Oceans	geometry
0	S.Atlantic	1	South Atlantic Ocean	POLYGON ((-67.26025728926088 -59.9309210526315...
1	N.Pacific	0	North Pacific Ocean	(POLYGON ((180 66.27034771241199, 180 0, 101.1...
2	Southern	3	Southern Ocean	POLYGON ((180 -60, 180 -90, -180 -90, -180 -60...
3	Arctic	2	Arctic Ocean	POLYGON ((-100.1196521436255 52.89103112710165...
4	Indian	5	Indian Ocean	POLYGON ((19.69705552221351 -59.94160091330382...

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));
```

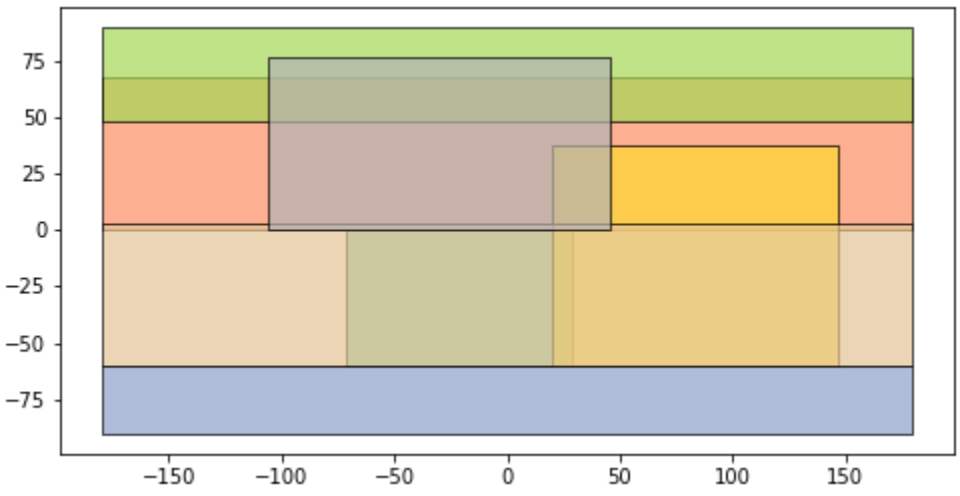


oceans.shp stores both Polygon and Multi-Polygon geometry types (a Polygon may also be viewed as a Multi-Polygon with 1 member). We can get at the geometry types and other geometry properties easily.

	minx	miny	maxx	maxy
0	-71.183612	-60.000000	28.736134	0.000000
1	-180.000000	0.000000	180.000000	67.479386
2	-180.000000	-90.000000	180.000000	-59.806846
3	-180.000000	47.660532	180.000000	90.000000
4	19.697056	-59.945004	146.991853	37.102940
5	-180.000000	-60.000000	180.000000	2.473291
6	-106.430148	0.000000	45.468236	76.644442

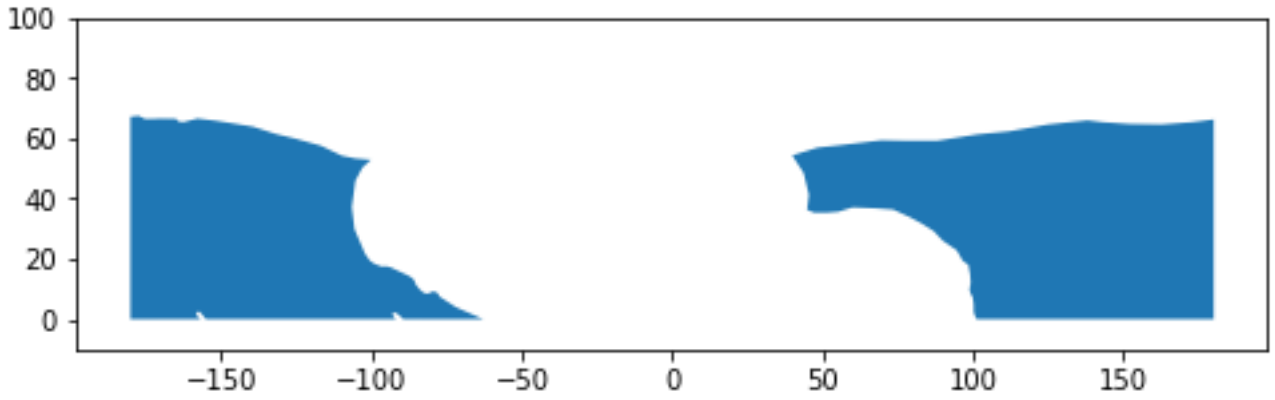
The envelope method returns the bounding box for each polygon.

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



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

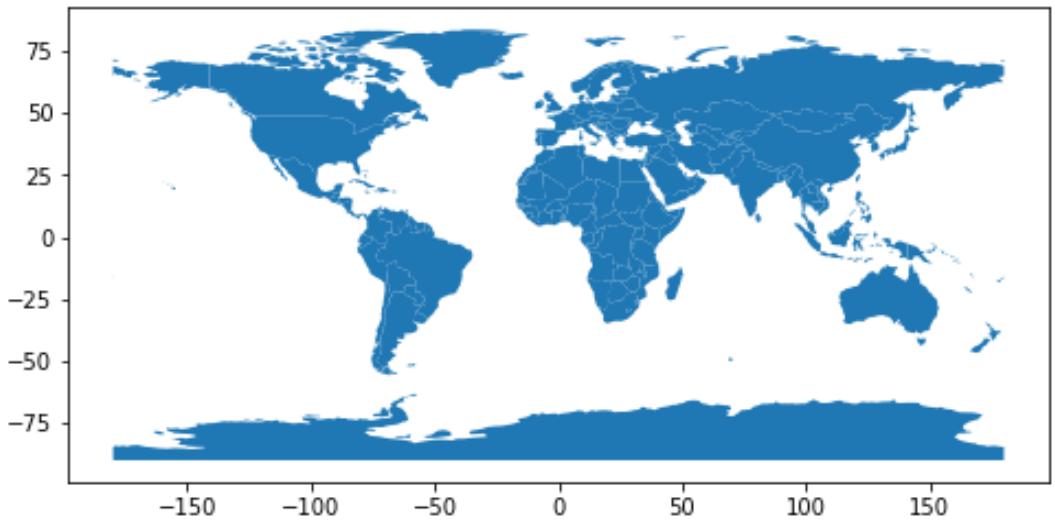
"*Natural Earth* is a public domain map dataset available at 1:10m, 1:50m, and 1:110 million scales. Featuring tightly integrated vector and raster data." A subset comes bundled with GeoPandas and is accessible from the gpd.datasets module. We'll use it as a helpful global base layer map.

```
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
world.head(2)
```

	pop_est	continent	name	iso_a3	gdp_md_est	geometry
0	28400000.0	Asia	Afghanistan	AFG	22270.0	POLYGON ((61.21081709172574 35.65007233330923,...
1	12799293.0	Africa	Angola	AGO	110300.0	(POLYGON ((16.32652835456705 -5.87747039146621...

Its CRS is also EPSG:4326:

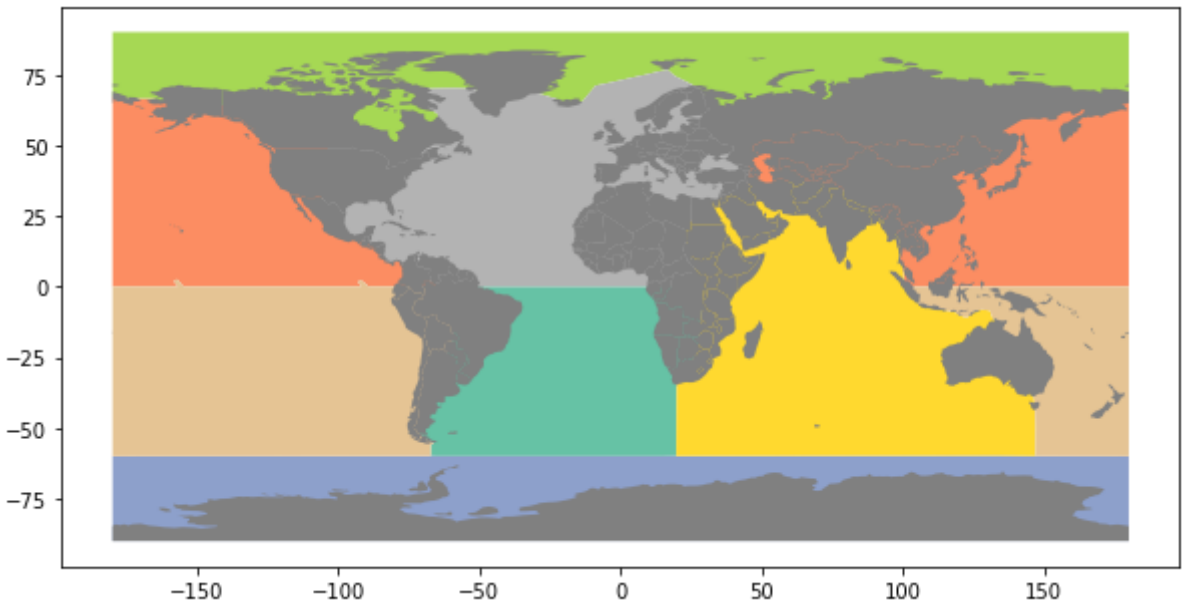
```
world.crs
    {"init": "epsg:4326"}
world.plot(figsize=(8, 8));
```



Map plot overlays: Plotting multiple spatial layers

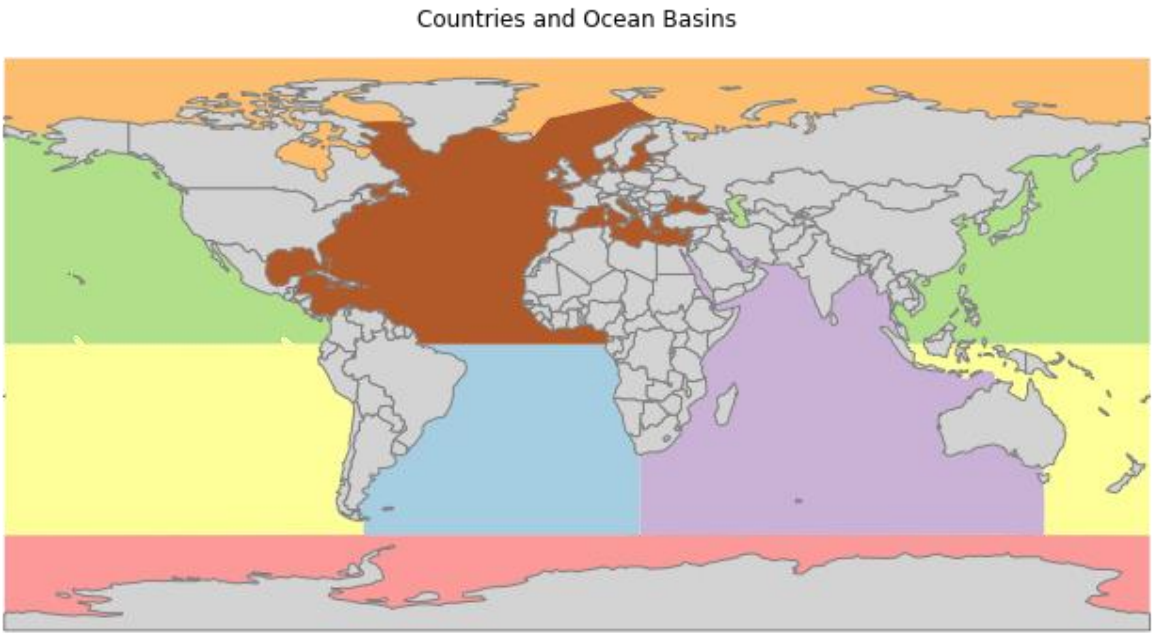
Here's a compact, quick way of using the GeoDataFrame plot method to overlay two GeoDataFrames while customizing the styles for each layer.

```
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');
```



5. 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"}
```

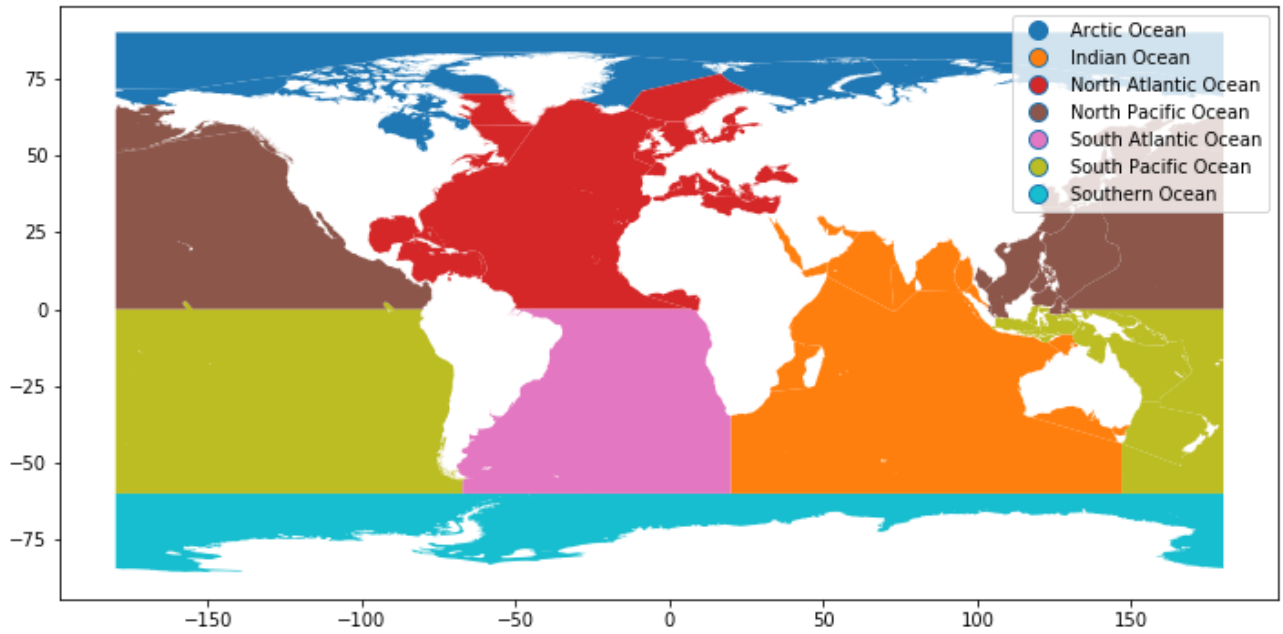
Let's take a look at the GeoDataFrame.

	gid	name	id	gazetteer	is_generic	oceans	geom
0	1	Inner Seas off the West Coast of Scotland	18	4283	False	North Atlantic Ocean	(POLYGON ((-6.496945454545455 58.0874909090909...
1	2	Mediterranean Sea - Western Basin	28A	4279	False	North Atlantic Ocean	(POLYGON ((12.4308 37.80325454545454, 12.41498...
2	3	Mediterranean Sea - Eastern Basin	28B	4280	False	North Atlantic Ocean	(POLYGON ((23.60853636363636 35.60874545454546...
3	4	Sea of Marmara	29	3369	False	North Atlantic Ocean	(POLYGON ((26.21790909090909 40.05290909090909...
4	5	Black Sea	30	3319	False	North Atlantic Ocean	(POLYGON ((29.04846363636364 41.25555454545454...

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));
```



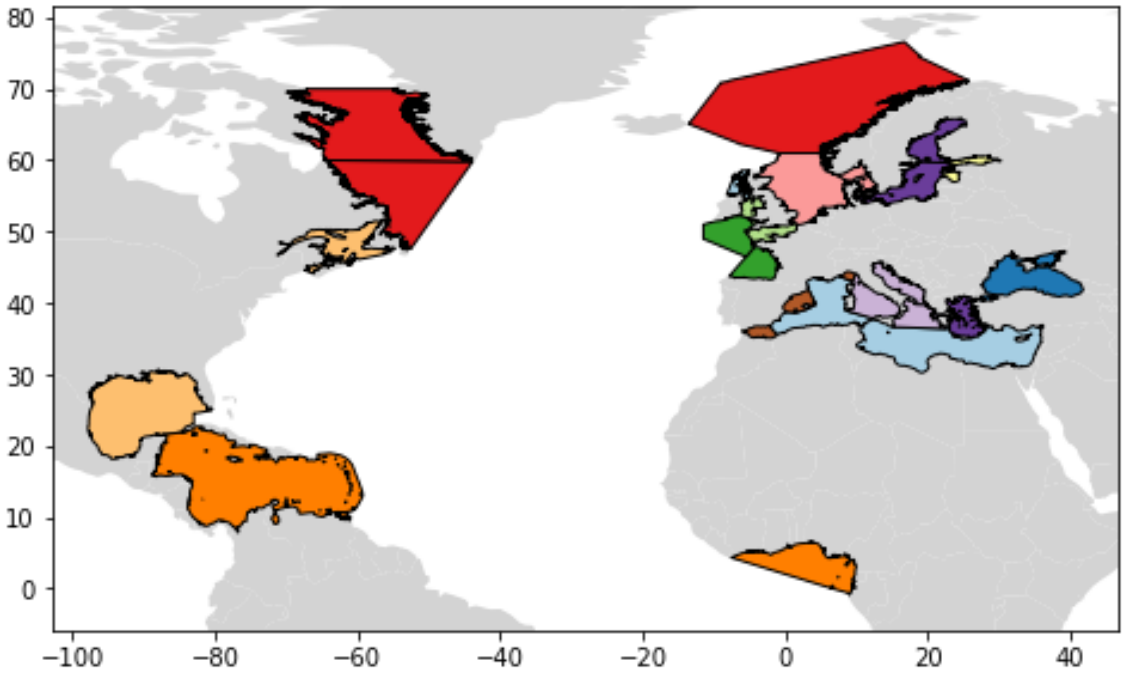
Additional plotting examples

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);
```