GeoPandas Documentation

Release 0.2.0.dev

Kelsey Jordahl

Sep 18, 2017

Contents

GeoPandas is an open source project to make working with geospatial data in python easier. GeoPandas extends the datatypes used by pandas to allow spatial operations on geometric types. Geometric operations are performed by shapely. Geopandas further depends on fiona for file access and descartes and matplotlib for plotting.

CHAPTER 1

Description

The goal of GeoPandas is to make working with geospatial data in python easier. It combines the capabilities of pandas and shapely, providing geospatial operations in pandas and a high-level interface to multiple geometries to shapely. GeoPandas enables you to easily do operations in python that would otherwise require a spatial database such as PostGIS.

Installation

Installing GeoPandas

To install the released version, you can use pip:

```
pip install geopandas
```

or you can install the conda package from the conda-forge channel:

```
conda install -c conda-forge geopandas
```

You may install the latest development version by cloning the *GitHub* repository and using the setup script:

```
git clone https://github.com/geopandas/geopandas.git
cd geopandas
pip install .
```

It is also possible to install the latest development version available on PyPI with *pip* by adding the `--pre` flag for pip 1.4 and later, or to use *pip* to install directly from the GitHub repository with:

```
pip install git+git://github.com/geopandas/geopandas.git
```

Dependencies

Installation via *conda* should also install all dependencies, but a complete list is as follows:

- numpy
- pandas (version 0.13 or later)
- shapely
- fiona
- six
- pyproj

Further, optional dependencies are:

- geopy 0.99 (optional; for geocoding)
- psycopg2 (optional; for PostGIS connection)
- rtree (optional; spatial index to improve performance)

For plotting, these additional packages may be used:

- matplotlib
- descartes
- pysal

These can be installed independently via the following set of commands:

```
conda install -c conda-forge fiona shapely pyproj rtree
conda install pandas
```

Data Structures

GeoPandas implements two main data structures, a `GeoSeries` and a `GeoDataFrame`. These are subclasses of pandas `Series` and `DataFrame`, respectively.

GeoSeries

A `GeoSeries` is essentially a vector where each entry in the vector is a set of shapes corresponding to one observation. An entry may consist of only one shape (like a single polygon) or multiple shapes that are meant to be thought of as one observation (like the many polygons that make up the State of Hawaii or a country like Indonesia).

geopandas has three basic classes of geometric objects (which are actually *shapely* objects):

- Points / Multi-Points
- Lines / Multi-Lines
- Polygons / Multi-Polygons

Note that all entries in a `GeoSeries` need not be of the same geometric type, although certain export operations will fail if this is not the case.

Overview of Attributes and Methods

The `GeoSeries` class implements nearly all of the attributes and methods of Shapely objects. When applied to a `GeoSeries`, they will apply elementwise to all geometries in the series. Binary operations can be applied between two `GeoSeries`, in which case the operation is carried out elementwise. The two series will be aligned by matching indices. Binary operations can also be applied to a single geometry, in which case the operation is carried out for each element of the series with that geometry. In either case, a `Series` or a `GeoSeries` will be returned, as appropriate.

A short summary of a few attributes and methods for GeoSeries is presented here, and a full list can be found in the *all attributes and methods page*. There is also a family of methods for creating new shapes by expanding existing shapes or applying set-theoretic operations like "union" described in *geometric manipulations*.

Attributes

- `area`: shape area (units of projection – see *projections*)
- `bounds`: tuple of max and min coordinates on each axis for each shape
- `total_bounds`: tuple of max and min coordinates on each axis for entire GeoSeries
- `geom_type`: type of geometry.
- `is_valid`: tests if coordinates make a shape that is reasonable geometric shape (according to this).

Basic Methods

- `distance(other)`: returns `Series` with minimum distance from each entry to `other`
- `centroid`: returns `GeoSeries` of centroids
- `representative_point()`: returns `GeoSeries` of points that are guaranteed to be within each geometry. It does **NOT** return centroids.
- `to_crs()`: change coordinate reference system. See *projections*
- `plot()`: plot `GeoSeries`. See *mapping*.

Relationship Tests

- `almost_equals(other)`: is shape almost the same as `other` (good when floating point precision issues make shapes slightly different)
- `contains(other)`: is shape contained within `other`
- `intersects(other)`: does shape intersect `other`

GeoDataFrame

A `GeoDataFrame` is a tabular data structure that contains a `GeoSeries`.

The most important property of a `GeoDataFrame` is that it always has one `GeoSeries` column that holds a special status. This `GeoSeries` is referred to as the `GeoDataFrame`'s "geometry". When a spatial method is applied to a `GeoDataFrame` (or a spatial attribute like `area` is called), this commands will always act on the "geometry" column.

The "geometry" column – no matter its name – can be accessed through the `geometry` attribute (`gdf.geometry`), and the name of the `geometry` column can be found by typing `gdf.geometry.name`.

A `GeoDataFrame` may also contain other columns with geometrical (shapely) objects, but only one column can be the active geometry at a time. To change which column is the active geometry column, use the `set_geometry` method.

An example using the `worlds` GeoDataFrame:

```
In [1]: world.head()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6e4c6b8ad2c1> in <module>()
----> 1 world.head()

NameError: name 'world' is not defined

#Plot countries
In [2]: world.plot();
```

Currently, the column named "borders" with country borders is the active geometry column:

```
In [3]: world.geometry.name
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-c347be3ed172> in <module>()
----> 1 world.geometry.name

NameError: name 'world' is not defined
```

Now, we create centroids and make it the geometry:

```
In [4]: world['centroid_column'] = world.centroid
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-342b4f803c99> in <module>()
----> 1 world['centroid_column'] = world.centroid

NameError: name 'world' is not defined

In [5]: world = world.set_geometry('centroid_column')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-c092184b7fb7> in <module>()
----> 1 world = world.set_geometry('centroid_column')
```

```
NameError: name 'world' is not defined

In [6]: world.plot();
```

Note: A `GeoDataFrame` keeps track of the active column by name, so if you rename the active geometry column, you must also reset the geometry:

```
gdf = gdf.rename(columns={'old_name': 'new_name'}).set_geometry('new_name')
```

Note 2: Somewhat confusingly, by default when you use the `read_file` command, the column containing spatial objects from the file is named "geometry" by default, and will be set as the active geometry column. However, despite using the same term for the name of the column and the name of the special attribute that keeps track of the active column, they are distinct. You can easily shift the active geometry column to a different `GeoSeries` with the `set_geometry` command. Further, `gdf.geometry` will always return the active geometry column, *not* the column named `geometry`. If you wish to call a column named "geometry", and a different column is the active geometry column, use `gdf['geometry']`, not `gdf.geometry`.

Attributes and Methods

Any of the attributes calls or methods described for a `GeoSeries` will work on a `GeoDataFrame` – effectively, they are just applied to the "geometry" `GeoSeries`.

However, `GeoDataFrames` also have a few extra methods for input and output which are described on the *Input and Output* page and for geocoding with are described in *Geocoding*.

Reading and Writing Files

Reading Spatial Data

geopandas can read almost any vector-based spatial data format including ESRI shapefile, GeoJSON files and more using the command:

```
gpd.read_file()
```

which returns a GeoDataFrame object. (This is possible because *geopandas* makes use of the great fiona library, which in turn makes use of a massive open-source program called GDAL/OGR designed to facilitate spatial data transformations).

Any arguments passed to `read_file()` after the file name will be passed directly to `fiona.open`, which does the actual data importation. In general, `read_file` is pretty smart and should do what you want without extra arguments, but for more help, type:

```
import fiona; help(fiona.open)
```

Among other things, one can explicitly set the driver (shapefile, GeoJSON) with the `driver` keyword, or pick a single layer from a multi-layered file with the `layer` keyword.

geopandas can also get data from a PostGIS database using the `read_postgis()` command.

Writing Spatial Data

GeoDataFrames can be exported to many different standard formats using the `GeoDataFrame.to_file()` method. For a full list of supported formats, type `import fiona; fiona.supported_drivers`.

Mapping Tools

geopandas provides a high-level interface to the `matplotlib` library for making maps. Mapping shapes is as easy as using the `plot()` method on a `GeoSeries` or `GeoDataFrame`.

```
# Examine country GeoDataFrame
In [1]: world.head()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6e4c6b8ad2c1> in <module>()
----> 1 world.head()

NameError: name 'world' is not defined

# Basic plot, random colors
In [2]: world.plot();
```

Note that in general, any options one can pass to pyplot in `matplotlib` (or style options that work for lines) can be passed to the `plot()` method.

Chloropleth Maps

geopandas makes it easy to create Chloropleth maps (maps where the color of each shape is based on the value of an associated variable). Simply use the plot command with the `column` argument set to the column whose values you want used to assign colors.

```
# Plot by GDP per capta
In [3]: world = world[(world.pop_est>0) & (world.name!="Antarctica")]
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-3d65aa1f3698> in <module>()
----> 1 world = world[(world.pop_est>0) & (world.name!="Antarctica")]

NameError: name 'world' is not defined

In [4]: world['gdp_per_cap'] = world.gdp_md_est / world.pop_est
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-b9fbf9f3897f> in <module>()
----> 1 world['gdp_per_cap'] = world.gdp_md_est / world.pop_est

NameError: name 'world' is not defined

In [5]: world.plot(column='gdp_per_cap');
```

Choosing colors

One can also modify the colors used by `plot` with the `cmap` option (for a full list of colormaps, see the matplotlib website):

```
In [6]: world.plot(column='gdp_per_cap', cmap='OrRd');
```

The way color maps are scaled can also be manipulated with the `scheme` option (if you have `pysal` installed, which can be accomplished via `conda install pysal`). By default, `scheme` is set to 'equal_intervals', but it can also be adjusted to any other pysal option, like 'quantiles', 'percentiles', etc.

```
In [7]: world.plot(column='gdp_per_cap', cmap='OrRd', scheme='quantiles');
```

Maps with Layers

There are two strategies for making a map with multiple layers – one more succinct, and one that is a littel more flexible.

Before combining maps, however, remember to always ensure they share a common CRS (so they will align).

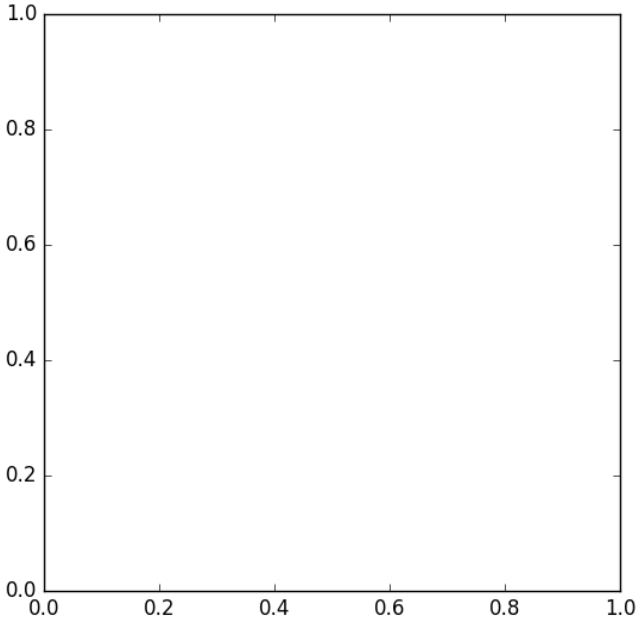
```
# Look at capitals
# Note use of standard `pyplot` line style options
In [8]: cities.plot(marker='*', color='green', markersize=5);

# Check crs
In [9]: cities = cities.to_crs(world.crs)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-31469edf5e1f> in <module>()
----> 1 cities = cities.to_crs(world.crs)

NameError: name 'cities' is not defined

# Now we can overlay over country outlines
# And yes, there are lots of island capitals
# apparently in the middle of the ocean!
```

Method 1

```
In [10]: base = world.plot(color='white')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-91e7c0d2b84e> in <module>()
----> 1 base = world.plot(color='white')

NameError: name 'world' is not defined

In [11]: cities.plot(ax=base, marker='o', color='red', markersize=5);
```

Method 2: Using matplotlib objects

```
In [12]: import matplotlib.pyplot as plt

In [13]: fig, ax = plt.subplots()

# set aspect to equal. This is done automatically
# when using *geopandas* plot on it's own, but not when
# working with pyplot directly.
In [14]: ax.set_aspect('equal')

In [15]: world.plot(ax=ax, color='white')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-9fc582f480d1> in <module>()
----> 1 world.plot(ax=ax, color='white')

NameError: name 'world' is not defined

In [16]: cities.plot(ax=ax, marker='o', color='red', markersize=5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-ec592ff81f19> in <module>()
----> 1 cities.plot(ax=ax, marker='o', color='red', markersize=5)

NameError: name 'cities' is not defined

In [17]: plt.show();
```



Other Resources

Links to jupyter Notebooks for different mapping tasks:

Making Heat Maps

Managing Projections

Coordinate Reference Systems

CRS are important because the geometric shapes in a GeoSeries or GeoDataFrame object are simply a collection of coordinates in an arbitrary space. A CRS tells Python how those coordinates related to places on the Earth.

CRS are referred to using codes called proj4 strings. You can find the codes for most commonly used projections from www.spatialreference.org or remotesensing.org.

The same CRS can often be referred to in many ways. For example, one of the most commonly used CRS is the WGS84 latitude-longitude projection. One *proj4* representation of this projection is: `"+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"`. But common projections can also be referred to by *EPSG* codes, so this same projection can also called using the *proj4* string `"+init=epsg:4326"`.

geopandas can accept lots of representations of CRS, including the *proj4* string itself (`"+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"`) or parameters broken out in a dictionary: `{'proj': 'latlong', 'ellps': 'WGS84', 'datum': 'WGS84', 'no_defs': True}`). In addition, some functions will take *EPSG* codes directly.

For reference, a few very common projections and their proj4 strings:

- WGS84 Latitude/Longitude: `"+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"` or `"+init=epsg:4326"`

- UTM Zones (North): `"+proj=utm +zone=33 +ellps=WGS84 +datum=WGS84 +units=m +no_defs"`
- UTM Zones (South): `"+proj=utm +zone=33 +ellps=WGS84 +datum=WGS84 +units=m +no_defs +south"`

Setting a Projection

There are two relevant operations for projections: setting a projection and re-projecting.

Setting a projection may be necessary when for some reason *geopandas* has coordinate data (x-y values), but no information about how those coordinates refer to locations in the real world. Setting a projection is how one tells *geopandas* how to interpret coordinates. If no CRS is set, *geopandas* geometry operations will still work, but coordinate transformations will not be possible and exported files may not be interpreted correctly by other software.

Be aware that **most of the time** you don't have to set a projection. Data loaded from a reputable source (using the `from_file()` command) *should* always include projection information. You can see an objects current CRS through the `crs` attribute: `my_geoseries.crs`.

From time to time, however, you may get data that does not include a projection. In this situation, you have to set the CRS so *geopandas* knows how to interpret the coordinates.

For example, if you convert a spreadsheet of latitudes and longitudes into a GeoSeries by hand, you would set the projection by assigning the WGS84 latitude-longitude CRS to the `crs` attribute:

```
my_geoseries.crs = {'init' :'epsg:4326'}
```

Re-Projecting

Re-projecting is the process of changing the representation of locations from one coordinate system to another. All projections of locations on the Earth into a two-dimensional plane are distortions, the projection that is best for your application may be different from the projection associated with the data you import. In these cases, data can be re-projected using the `to_crs` command:

```
# Check original projection
# (it's Platte Carre! x-y are long and lat)
In [1]: world.crs
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6dccfe34da9b> in <module>()
----> 1 world.crs

NameError: name 'world' is not defined

# Visualize
In [2]: world.plot();

# Reproject to Mercator (after dropping Antartica)
In [3]: world = world[(world.name != "Antarctica") & (world.name != "Fr. S. Antarctic␣
↪Lands")]
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-0675a59ff78c> in <module>()
----> 1 world = world[(world.name != "Antarctica") & (world.name != "Fr. S. Antarctic␣
↪Lands")]
```

```
NameError: name 'world' is not defined

In [4]: world = world.to_crs({'init': 'epsg:3395'}) # world.to_crs(epsg=3395) would␣
↪also work
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪--------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-5a29e9b4f4c0> in <module>()
----> 1 world = world.to_crs({'init': 'epsg:3395'}) # world.to_crs(epsg=3395) would␣
↪also work

NameError: name 'world' is not defined

In [5]: world.plot();
```

Geometric Manipulations

Set-theoretic Methods

`GeoSeries.`**boundary**

Returns a `GeoSeries` of lower dimensional objects representing each geometries's set-theoretic *boundary*.

`GeoSeries.`**difference**(*other*)

Returns a `GeoSeries` of the points in each geometry that are not in the *other* object.

GeoSeries.**intersection**(*other*)
Returns a GeoSeries of the intersection of each object with the *other* geometric object.

GeoSeries.**symmetric_difference**(*other*)
Returns a GeoSeries of the points in each object not in the *other* geometric object, and the points in the *other* not in this object.

GeoSeries.**union**(*other*)
Returns a GeoSeries of the union of points from each object and the *other* geometric object.

GeoSeries.**unary_union**
Return a geometry containing the union of all geometries in the GeoSeries.

Constructive Methods

GeoSeries.**buffer**(*distance*, *resolution=16*)
Returns a GeoSeries of geometries representing all points within a given *distance* of each geometric object.

GeoSeries.**convex_hull**
Returns a GeoSeries of geometries representing the smallest convex *Polygon* containing all the points in each object unless the number of points in the object is less than three. For two points, the convex hull collapses to a *LineString*; for 1, a *Point*.

GeoSeries.**envelope**
Returns a GeoSeries of geometries representing the point or smallest rectangular polygon (with sides parallel to the coordinate axes) that contains each object.

GeoSeries.**simplify**(*tolerance*, *preserve_topology=True*)
Returns a GeoSeries containing a simplified representation of each object.

Affine transformations

GeoSeries.**rotate**(*self*, *angle*, *origin='center'*, *use_radians=False*)
Rotate the coordinates of the GeoSeries.

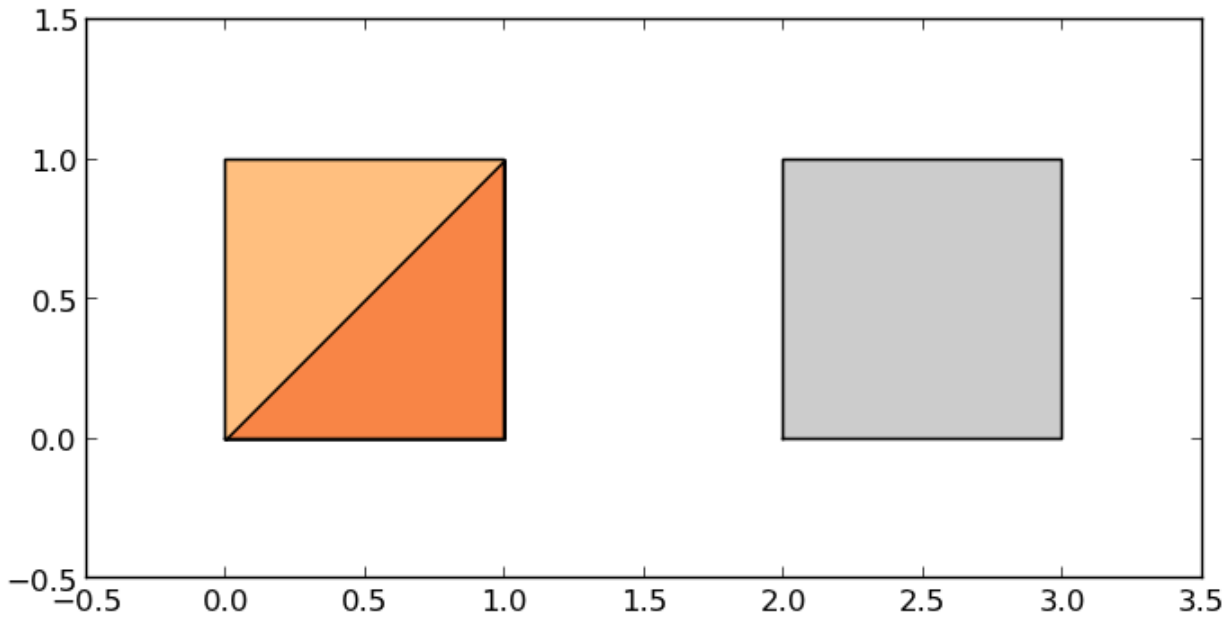
GeoSeries.**scale**(*self*, *xfact=1.0*, *yfact=1.0*, *zfact=1.0*, *origin='center'*)
Scale the geometries of the GeoSeries along each (x, y, z) dimensio.

GeoSeries.**skew**(*self*, *angle*, *origin='center'*, *use_radians=False*)
Shear/Skew the geometries of the GeoSeries by angles along x and y dimensions.

GeoSeries.**translate**(*self*, *angle*, *origin='center'*, *use_radians=False*)
Shift the coordinates of the GeoSeries.

Aggregating methods

```
>>> p1 = Polygon([(0, 0), (1, 0), (1, 1)])
>>> p2 = Polygon([(0, 0), (1, 0), (1, 1), (0, 1)])
>>> p3 = Polygon([(2, 0), (3, 0), (3, 1), (2, 1)])
>>> g = GeoSeries([p1, p2, p3])
>>> g
0    POLYGON ((0.0000000000000000 0.000000000000000...
1    POLYGON ((0.0000000000000000 0.000000000000000...
2    POLYGON ((2.0000000000000000 0.000000000000000...
dtype: object
```

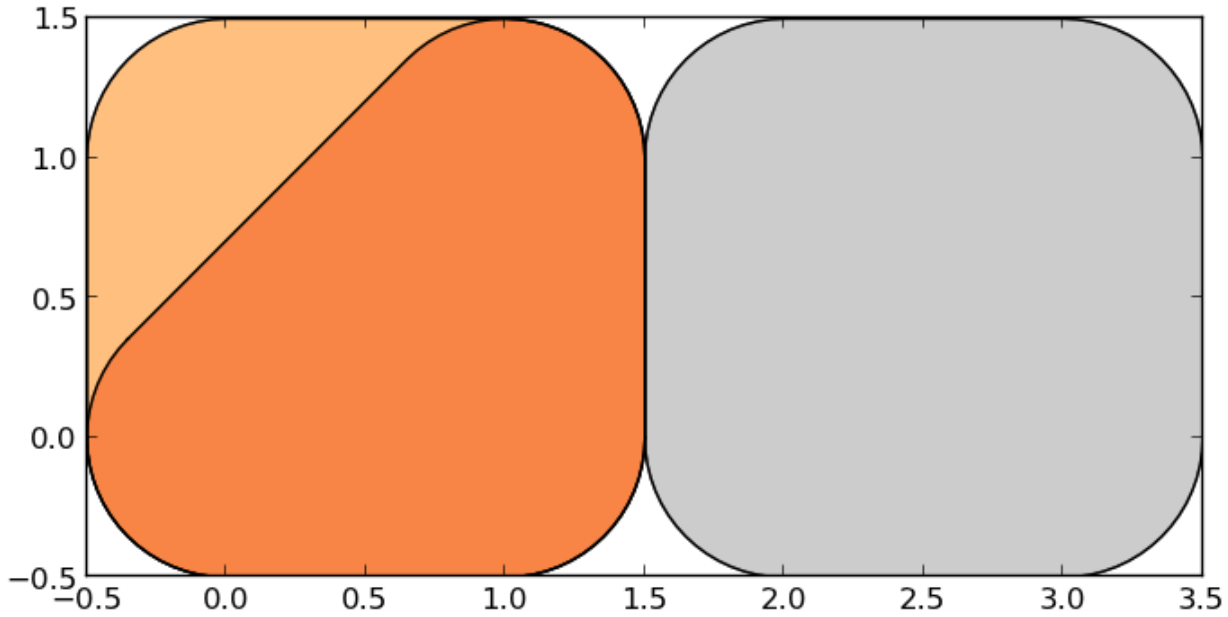


Some geographic operations return normal pandas object. The `area` property of a `GeoSeries` will return a `pandas.Series` containing the area of each item in the `GeoSeries`:

```
>>> print g.area
0    0.5
1    1.0
2    1.0
dtype: float64
```

Other operations return GeoPandas objects:

```
>>> g.buffer(0.5)
Out[15]:
0    POLYGON ((-0.3535533905932737 0.35355339059327...
1    POLYGON ((-0.5000000000000000 0.00000000000000...
2    POLYGON ((1.5000000000000000 0.000000000000000...
dtype: object
```

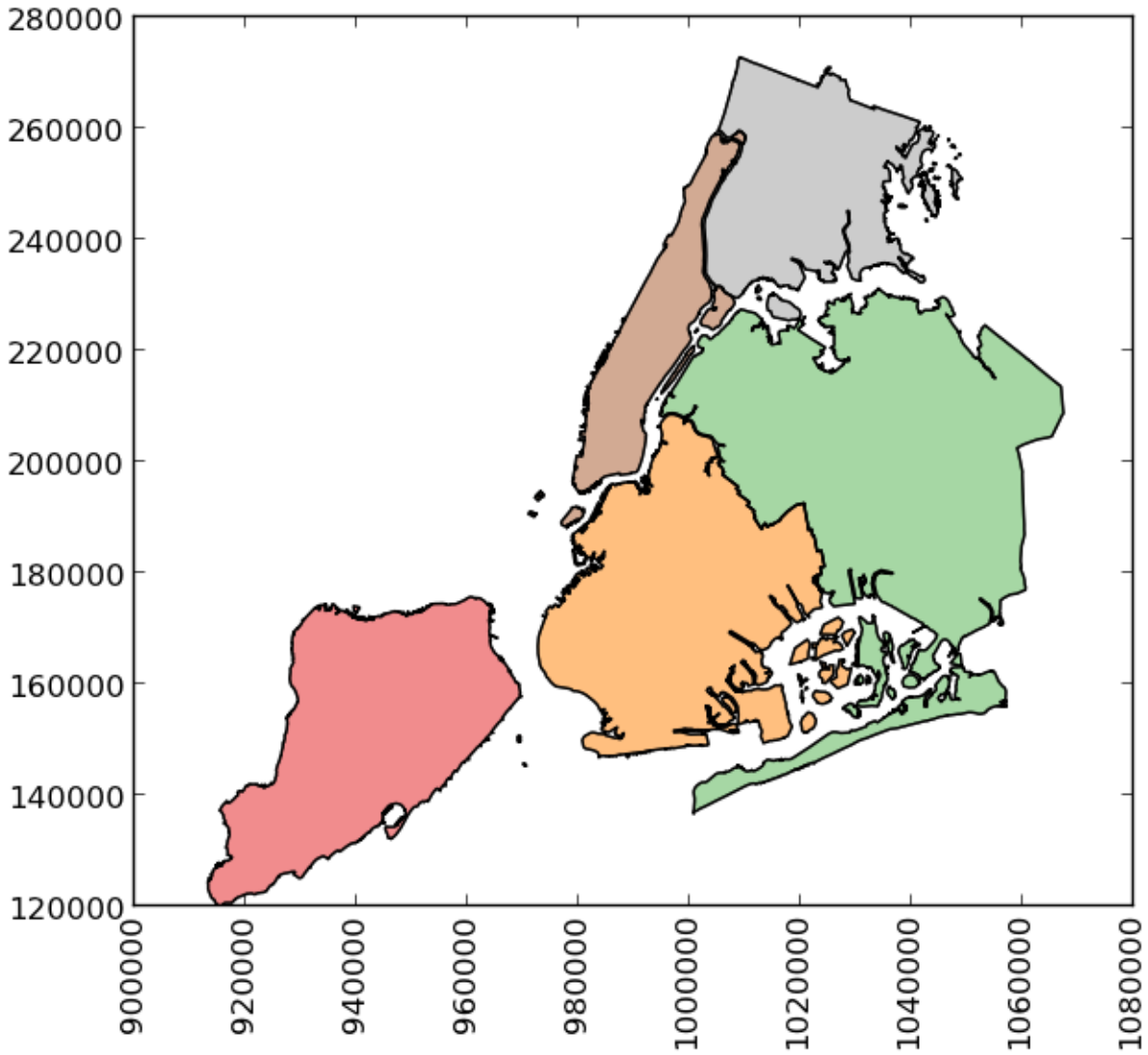


GeoPandas objects also know how to plot themselves. GeoPandas uses descartes to generate a matplotlib plot. To generate a plot of our GeoSeries, use:

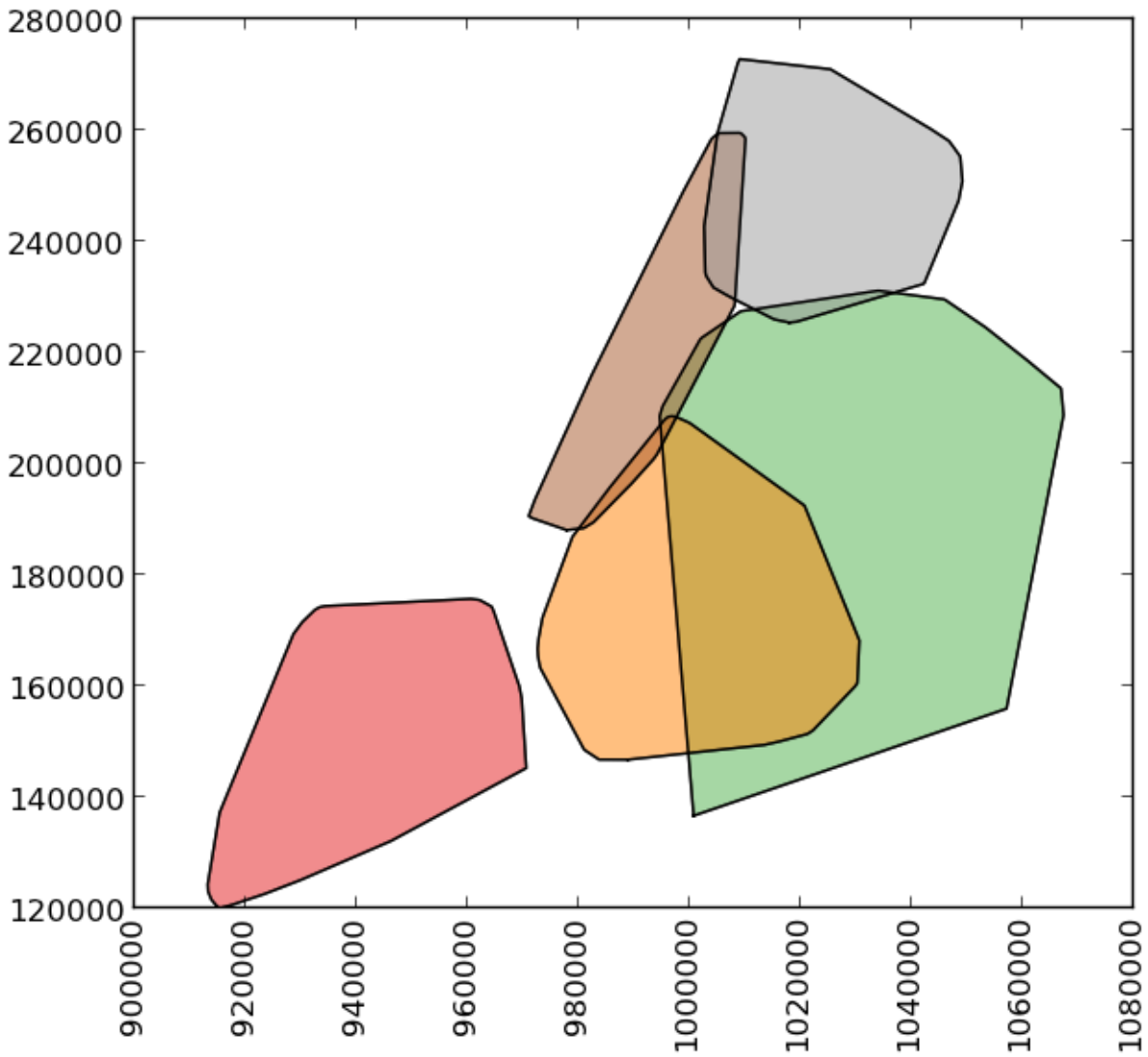
```
>>> g.plot()
```

GeoPandas also implements alternate constructors that can read any data format recognized by fiona. To read a file containing the boroughs of New York City:

```
>>> boros = GeoDataFrame.from_file('nybb.shp')
>>> boros.set_index('BoroCode', inplace=True)
>>> boros.sort()
>>> boros
               BoroName    Shape_Area     Shape_Leng  \
BoroCode
1             Manhattan  6.364422e+08  358532.956418
2                 Bronx  1.186804e+09  464517.890553
3              Brooklyn  1.959432e+09  726568.946340
4                Queens  3.049947e+09  861038.479299
5         Staten Island  1.623853e+09  330385.036974

                                                  geometry
BoroCode
1          (POLYGON ((981219.0557861328125000 188655.3157...
2          (POLYGON ((1012821.8057861328125000 229228.264...
3          (POLYGON ((1021176.4790039062500000 151374.796...
4          (POLYGON ((1029606.0765991210937500 156073.814...
5          (POLYGON ((970217.0223999023437500 145643.3322...
```




```
>>> boros['geometry'].convex_hull
0    POLYGON ((915517.6877458114176989 120121.88125...
1    POLYGON ((1000721.5317993164062500 136681.7761...
2    POLYGON ((988872.8212280273437500 146772.03179...
3    POLYGON ((977855.4451904296875000 188082.32238...
4    POLYGON ((1017949.9776000976562500 225426.8845...
dtype: object
```



To demonstrate a more complex operation, we'll generate a `GeoSeries` containing 2000 random points:

```
>>> from shapely.geometry import Point
>>> xmin, xmax, ymin, ymax = 900000, 1080000, 120000, 280000
>>> xc = (xmax - xmin) * np.random.random(2000) + xmin
>>> yc = (ymax - ymin) * np.random.random(2000) + ymin
>>> pts = GeoSeries([Point(x, y) for x, y in zip(xc, yc)])
```

Now draw a circle with fixed radius around each point:

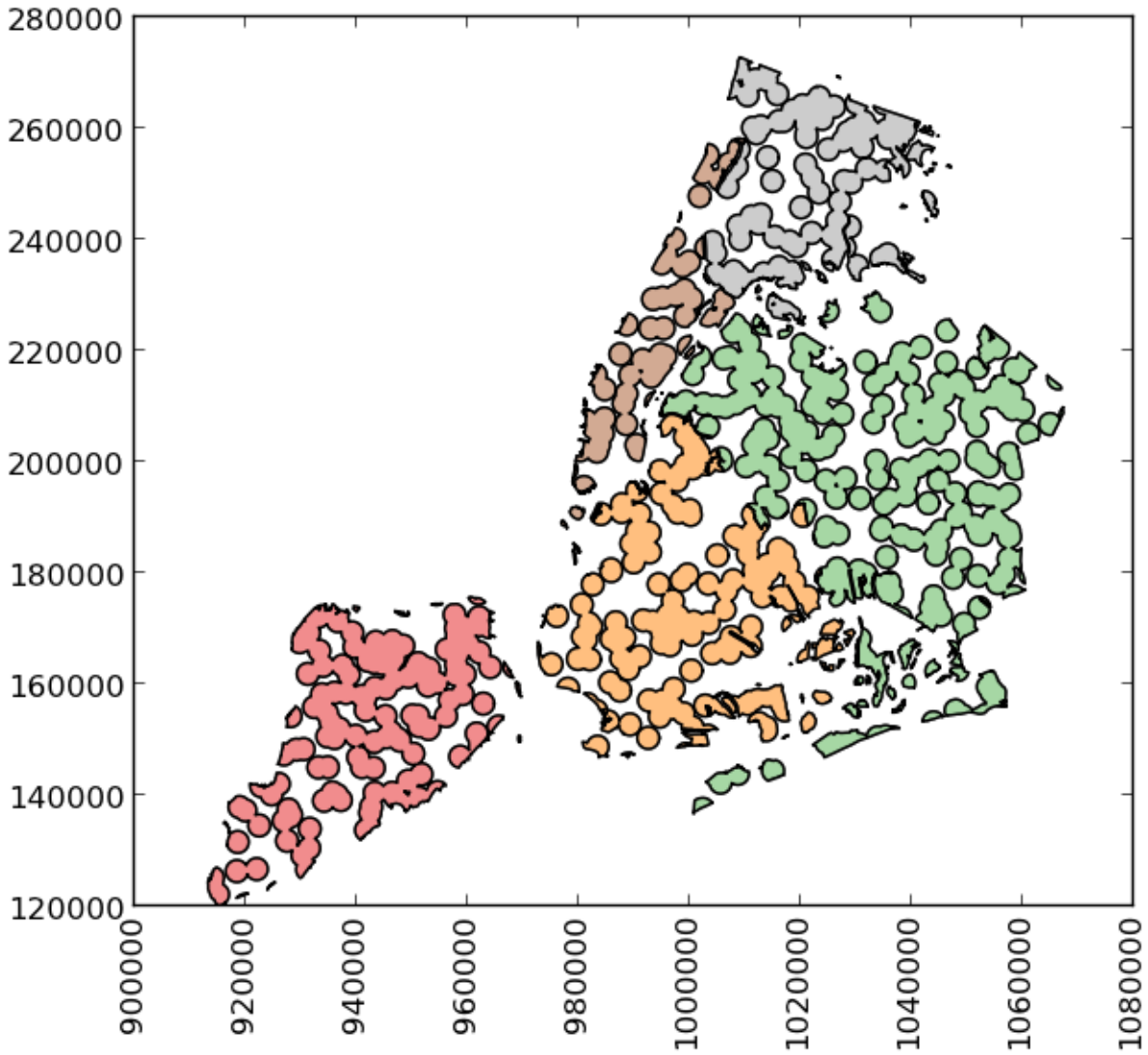
```
>>> circles = pts.buffer(2000)
```

We can collapse these circles into a single shapely MultiPolygon geometry with

```
>>> mp = circles.unary_union
```

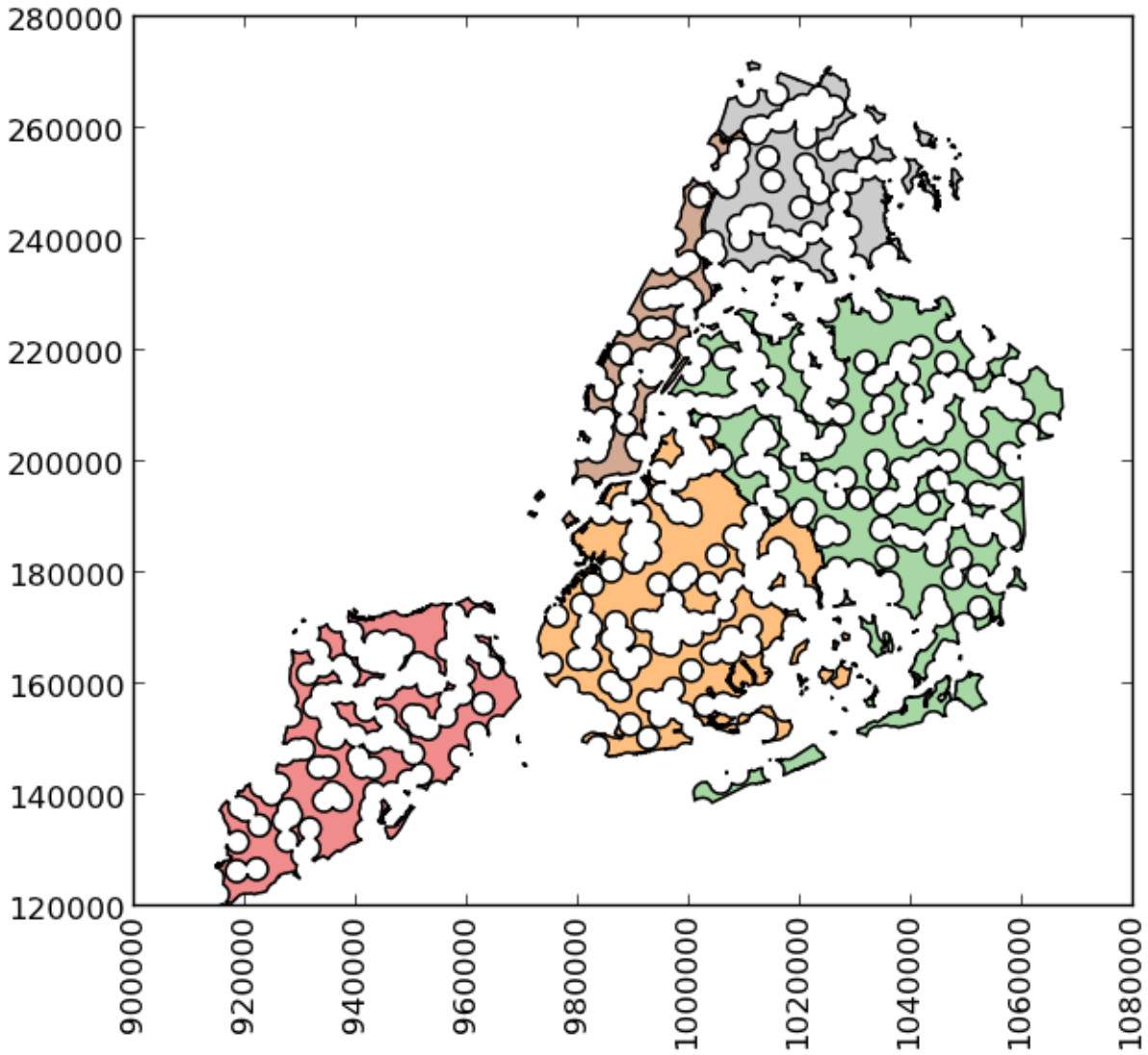
To extract the part of this geometry contained in each borough, we can just use:

```
>>> holes = boros['geometry'].intersection(mp)
```



and to get the area outside of the holes:

```
>>> boros_with_holes = boros['geometry'].difference(mp)
```



Note that this can be simplified a bit, since `geometry` is available as an attribute on a `GeoDataFrame`, and the `intersection` and `difference` methods are implemented with the "&" and "-" operators, respectively. For example, the latter could have been expressed simply as `boros.geometry - mp`.

It's easy to do things like calculate the fractional area in each borough that are in the holes:

```
>>> holes.area / boros.geometry.area
BoroCode
1           0.602015
2           0.523457
3           0.585901
4           0.577020
5           0.559507
dtype: float64
```

Merging Data

Attribute Joins

[TO BE COMPLETED – EXAMPLES OF JOINING GDF WITH PANDAS DATAFRAME]

Spatial Joins

[TO BE COMPLETED – EXAMPLES OF SPATIAL JOINS]

Geocoding

[TO BE COMPLETED]

`geopandas.geocode.`**`geocode`**(*strings*, *provider='googlev3'*, ***kwargs*)

Geocode a list of strings and return a GeoDataFrame containing the resulting points in its `geometry` column. Available `provider``s include ``googlev3`, `bing`, `google`, `yahoo`, `mapquest`, and `openmapquest`. `**kwargs` will be passed as parameters to the appropriate geocoder.

Requires **'geopy'_**. Please consult the Terms of Service for the chosen provider.

Reference

The following Shapely methods and attributes are available on `GeoSeries` objects:

`GeoSeries.`**`area`**

Returns a `Series` containing the area of each geometry in the `GeoSeries`.

`GeoSeries.`**`bounds`**

Returns a `DataFrame` with columns `minx`, `miny`, `maxx`, `maxy` values containing the bounds for each geometry. (see `GeoSeries.total_bounds` for the limits of the entire series).

`GeoSeries.`**`length`**

Returns a `Series` containing the length of each geometry.

`GeoSeries.`**`geom_type`**

Returns a `Series` of strings specifying the *Geometry Type* of each object.

`GeoSeries.`**`distance`**(*other*)

Returns a `Series` containing the minimum distance to the *other* `GeoSeries` (elementwise) or geometric object.

`GeoSeries.`**`representative_point`**`()`

Returns a `GeoSeries` of (cheaply computed) points that are guaranteed to be within each geometry.

`GeoSeries.`**`exterior`**

Returns a `GeoSeries` of LinearRings representing the outer boundary of each polygon in the GeoSeries. (Applies to GeoSeries containing only Polygons).

`GeoSeries.`**`interiors`**

Returns a `GeoSeries` of InteriorRingSequences representing the inner rings of each polygon in the GeoSeries. (Applies to GeoSeries containing only Polygons).

Unary Predicates

GeoSeries.**is_empty**
Returns a Series of dtype('bool') with value True for empty geometries.

GeoSeries.**is_ring**
Returns a Series of dtype('bool') with value True for features that are closed.

GeoSeries.**is_simple**
Returns a Series of dtype('bool') with value True for geometries that do not cross themselves (meaningful only for *LineStrings* and *LinearRings*).

GeoSeries.**is_valid**
Returns a Series of dtype('bool') with value True for geometries that are valid.

Binary Predicates

GeoSeries.**almost_equals**(*other*[, *decimal=6*])
Returns a Series of dtype('bool') with value True if each object is approximately equal to the *other* at all points to specified *decimal* place precision. (See also *equals()*)

GeoSeries.**contains**(*other*)
Returns a Series of dtype('bool') with value True if each object's *interior* contains the *boundary* and *interior* of the other object and their boundaries do not touch at all.

GeoSeries.**crosses**(*other*)
Returns a Series of dtype('bool') with value True if the *interior* of each object intersects the *interior* of the other but does not contain it, and the dimension of the intersection is less than the dimension of the one or the other.

GeoSeries.**disjoint**(*other*)
Returns a Series of dtype('bool') with value True if the *boundary* and *interior* of each object does not intersect at all with those of the other.

GeoSeries.**equals**(*other*)
Returns a Series of dtype('bool') with value True if if the set-theoretic *boundary*, *interior*, and *exterior* of each object coincides with those of the other.

GeoSeries.**intersects**(*other*)
Returns a Series of dtype('bool') with value True if if the *boundary* and *interior* of each object intersects in any way with those of the other.

GeoSeries.**touches**(*other*)
Returns a Series of dtype('bool') with value True if the objects have at least one point in common and their interiors do not intersect with any part of the other.

GeoSeries.**within**(*other*)
Returns a Series of dtype('bool') with value True if each object's *boundary* and *interior* intersect only with the *interior* of the other (not its *boundary* or *exterior*). (Inverse of *contains()*)

Set-theoretic Methods

GeoSeries.**boundary**
Returns a GeoSeries of lower dimensional objects representing each geometries's set-theoretic *boundary*.

GeoSeries.**centroid**
Returns a GeoSeries of points for each geometric centroid.

GeoSeries.**difference**(*other*)
Returns a GeoSeries of the points in each geometry that are not in the *other* object.

GeoSeries.**intersection**(*other*)
Returns a GeoSeries of the intersection of each object with the *other* geometric object.

GeoSeries.**symmetric_difference**(*other*)
Returns a GeoSeries of the points in each object not in the *other* geometric object, and the points in the *other* not in this object.

GeoSeries.**union**(*other*)
Returns a GeoSeries of the union of points from each object and the *other* geometric object.

Constructive Methods

GeoSeries.**buffer**(*distance*, *resolution=16*)
Returns a GeoSeries of geometries representing all points within a given *distance* of each geometric object.

GeoSeries.**convex_hull**
Returns a GeoSeries of geometries representing the smallest convex *Polygon* containing all the points in each object unless the number of points in the object is less than three. For two points, the convex hull collapses to a *LineString*; for 1, a *Point*.

GeoSeries.**envelope**
Returns a GeoSeries of geometries representing the point or smallest rectangular polygon (with sides parallel to the coordinate axes) that contains each object.

GeoSeries.**simplify**(*tolerance*, *preserve_topology=True*)
Returns a GeoSeries containing a simplified representation of each object.

Affine transformations

GeoSeries.**rotate**(*self*, *angle*, *origin='center'*, *use_radians=False*)
Rotate the coordinates of the GeoSeries.

GeoSeries.**scale**(*self*, *xfact=1.0*, *yfact=1.0*, *zfact=1.0*, *origin='center'*)
Scale the geometries of the GeoSeries along each (x, y, z) dimensio.

GeoSeries.**skew**(*self*, *angle*, *origin='center'*, *use_radians=False*)
Shear/Skew the geometries of the GeoSeries by angles along x and y dimensions.

GeoSeries.**translate**(*self*, *angle*, *origin='center'*, *use_radians=False*)
Shift the coordinates of the GeoSeries.

Aggregating methods

GeoSeries.**unary_union**
Return a geometry containing the union of all geometries in the GeoSeries.

Additionally, the following methods are implemented:

GeoSeries.**from_file**()
Load a GeoSeries from a file from any format recognized by **'fiona'_**.

GeoSeries.**to_crs**(*crs=None*, *epsg=None*)
Transform all geometries in a GeoSeries to a different coordinate reference system. The crs attribute on the current GeoSeries must be set. Either crs in dictionary form or an EPSG code may be specified for output.

This method will transform all points in all objects. It has no notion or projecting entire geometries. All segments joining points are assumed to be lines in the current projection, not geodesics. Objects crossing the dateline (or other projection boundary) will have undesirable behavior.

GeoSeries.**plot**(*colormap='Set1'*, *alpha=0.5*, *axes=None*)
Generate a plot of the geometries in the GeoSeries. colormap can be any recognized by matplotlib, but discrete colormaps such as Accent, Dark2, Paired, Pastel1, Pastel2, Set1, Set2, or Set3 are recommended. Wraps the plot_series() function.

GeoSeries.**total_bounds**
: Returns a tuple containing `minx`, `miny`, `maxx`, `maxy` values for the bounds of the series as a whole. See `GeoSeries.bounds` for the bounds of the geometries contained in the series.

GeoSeries.**__geo_interface__**
: Implements the 'geo_interface'_. Returns a python data structure to represent the `GeoSeries` as a GeoJSON-like `FeatureCollection`. Note that the features will have an empty `properties` dict as they don't have associated attributes (geometry only).

Methods of pandas `Series` objects are also available, although not all are applicable to geometric objects and some may return a `Series` rather than a `GeoSeries` result. The methods `copy()`, `align()`, `isnull()` and `fillna()` have been implemented specifically for `GeoSeries` and are expected to work correctly.

GeoDataFrame

A `GeoDataFrame` is a tablular data structure that contains a column called `geometry` which contains a *GeoSeries*'.

Currently, the following methods are implemented for a `GeoDataFrame`:

classmethod `GeoDataFrame.`**from_file**(*filename*, ***kwargs*)
: Load a `GeoDataFrame` from a file from any format recognized by 'fiona'_. See `read_file()`.

classmethod `GeoDataFrame.`**from_postgis**(*sql*, *con*, *geom_col='geom'*, *crs=None*, *index_col=None*, *coerce_float=True*, *params=None*)
: Load a `GeoDataFrame` from a file from a PostGIS database. See `read_postgis()`.

`GeoSeries.`**to_crs**(*crs=None*, *epsg=None*, *inplace=False*)
: Transform all geometries in the `geometry` column of a GeoDataFrame to a different coordinate reference system. The `crs` attribute on the current GeoSeries must be set. Either `crs` in dictionary form or an EPSG code may be specified for output. If `inplace=True` the geometry column will be replaced in the current dataframe, otherwise a new GeoDataFrame will be returned.

  This method will transform all points in all objects. It has no notion or projecting entire geometries. All segments joining points are assumed to be lines in the current projection, not geodesics. Objects crossing the dateline (or other projection boundary) will have undesirable behavior.

`GeoSeries.`**to_file**(*filename*, *driver="ESRI Shapefile"*, ***kwargs*)
: Write the `GeoDataFrame` to a file. By default, an ESRI shapefile is written, but any OGR data source supported by Fiona can be written. `**kwargs` are passed to the Fiona driver.

`GeoSeries.`**to_json**(***kwargs*)
: Returns a GeoJSON representation of the `GeoDataFrame` as a string.

`GeoDataFrame.`**plot**(*column=None*, *colormap=None*, *alpha=0.5*, *categorical=False*, *legend=False*, *axes=None*)
: Generate a plot of the geometries in the `GeoDataFrame`. If the `column` parameter is given, colors plot according to values in that column, otherwise calls `GeoSeries.plot()` on the `geometry` column. Wraps the `plot_dataframe()` function.

`GeoDataFrame.`**__geo_interface__**
: Implements the 'geo_interface'_. Returns a python data structure to represent the `GeoDataFrame` as a GeoJSON-like `FeatureCollection`.

All pandas `DataFrame` methods are also available, although they may not operate in a meaningful way on the `geometry` column and may not return a `GeoDataFrame` result even when it would be appropriate to do so.

Contributing to GeoPandas

(Contribution guidelines largely copied from pandas)

Overview

Contributions to GeoPandas are very welcome. They are likely to be accepted more quickly if they follow these guidelines.

At this stage of GeoPandas development, the priorities are to define a simple, usable, and stable API and to have clean, maintainable, readable code. Performance matters, but not at the expense of those goals.

In general, GeoPandas follows the conventions of the pandas project where applicable.

In particular, when submitting a pull request:

- All existing tests should pass. Please make sure that the test suite passes, both locally and on Travis CI. Status on Travis will be visible on a pull request. If you want to enable Travis CI on your own fork, please read the pandas guidelines link above or the getting started docs.
- New functionality should include tests. Please write reasonable tests for your code and make sure that they pass on your pull request.
- Classes, methods, functions, etc. should have docstrings. The first line of a docstring should be a standalone summary. Parameters and return values should be ducumented explicitly.
- GeoPandas supports python 2 (2.6+) and python 3 (3.2+) with a single code base. Use modern python idioms when possible that are compatibile with both major versions, and use the six library where helpful to smooth over the differences. Use `from __future__ import` statements where appropriate. Test code locally in both python 2 and python 3 when possible (all supported versions will be automatically tested on Travis CI).
- Follow PEP 8 when possible.
- Imports should be grouped with standard library imports first, 3rd-party libraries next, and geopandas imports third. Within each grouping, imports should be alphabetized. Always use absolute imports when possible, and explicit relative imports for local imports when necessary in tests.

Seven Steps for Contributing

There are seven basic steps to contributing to *geopandas*:

1. Fork the *geopandas* git repository
2. Create a development environment
3. Install *geopandas* dependencies
4. Make a `development` build of *geopandas*
5. Make changes to code and add tests
6. Update the documentation
7. Submit a Pull Request

Each of these 7 steps is detailed below.

1) Forking the *geopandas* repository using Git

To the new user, working with Git is one of the more daunting aspects of contributing to *geopandas**. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:

- Software Carpentry's Git Tutorial
- Atlassian
- the GitHub help pages.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the geopandas project page and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone git@github.com:your-user-name/geopandas.git geopandas-yourname
cd geopandas-yourname
git remote add upstream git://github.com/geopandas/geopandas.git
```

This creates the directory *geopandas-yourname* and connects your repository to the upstream (main project) *geopandas* repository.

The testing suite will run automatically on Travis-CI once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then Travis-CI needs to be hooked up to your GitHub repository. Instructions for doing so are here.

Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *geopandas*. You can have many shiny-new-features and switch in between them using the git checkout command.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the latest geopandas git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating.

2) Creating a development environment

A development environment is a virtual space where you can keep an independent installation of *geopandas*. This makes it easy to keep both a stable version of python in one place you use for work, and a development version (which you may break while playing with code) in another.

An easy way to create a *geopandas* development environment is as follows:

- Install either Anaconda or miniconda
- Make sure that you have *cloned the repository*
- `cd` to the *geopandas** source directory

Tell conda to create a new environment, named `geopandas_dev`, or any other name you would like for this environment, by running:

```
conda create -n geopandas_dev
```

For a python 3 environment:

```
conda create -n geopandas_dev python=3.4
```

This will create the new environment, and not touch any of your existing environments, nor any existing python installation.

To work in this environment, Windows users should `activate` it as follows:

```
activate geopandas_dev
```

Mac OSX and Linux users should use:

```
source activate geopandas_dev
```

You will then see a confirmation message to indicate you are in the new development environment.

To view your environments:

```
conda info -e
```

To return to you home root environment:

```
deactivate
```

See the full conda docs here.

At this point you can easily do a *development* install, as detailed in the next sections.

3) Installing Dependencies

To run *geopandas* in an development environment, you must first install *geopandas*'s dependencies. We suggest doing so using the following commands (executed after your development environment has been activated):

```
conda install -c conda-forge fiona shapely pyproj rtree
conda install pandas
```

This should install all necessary dependencies.

4) Making a development build

Once dependencies are in place, make an in-place build by navigating to the git clone of the *geopandas* repository and running:

```
python setup.py develop
```

5) Making changes and writing tests

geopandas is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). This development process "relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test." So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to *geopandas*. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

Like many packages, *geopandas* uses the Nose testing system and the convenient extensions in numpy.testing.

Writing tests

All tests should go into the `tests` directory. This folder contains many current examples of tests, and we suggest looking to these for inspiration.

The `.util` module has some special `assert` functions that make it easier to make statements about whether GeoSeries or GeoDataFrame objects are equivalent. The easiest way to verify that your code is correct is to explicitly construct the result you expect, then compare the actual result to the expected correct result, using eg the function `assert_geoseries_equal`.

Running the test suite

The tests can then be run directly inside your Git clone (without having to install *geopandas*) by typing:

```
nosetests -v
```

6) Updating the Documentation

geopandas documentation resides in the *doc* folder. Changes to the docs are make by modifying the appropriate file in the *source* folder within *doc*. *geopandas* docs us reStructuredText syntax, which is explained here and the docstrings follow the Numpy Docstring standard.

Once you have made your changes, you can build the docs by navigating to the *doc* folder and typing:

```
make html
```

The resulting html pages will be located in *doc/build/html*.

7) Submitting a Pull Request

Once you've made changes and pushed them to your forked repository, you then submit a pull request to have them integrated into the *geopandas* code base.

You can find a pull request (or PR) tutorial in the GitHub's Help Docs.

About

Known issues

- The `geopy` API has changed significantly over recent versions, `geopy 1.10.0` is currently supported.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

I

L

P

R

S

T

U

W