

GeoPandas 1.0.1

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 [pyogrio](#) for file access and [matplotlib](#) for plotting.

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.

[Getting started](#)[Documentation](#)[About GeoPandas](#)[Community](#)

Useful links

[Binary Installers \(PyPI\)](#) | [Source Repository \(GitHub\)](#) | [Issues & Ideas](#) | [Q&A Support](#)

     DOI [10.5281/zenodo.12625316](#)

powered by 

Supported by

The GeoPandas project uses an [open governance model](#) and is fiscally sponsored by [NumFOCUS](#). Consider making a [tax-deductible donation](#) to help the project pay for developer time, professional services, travel, workshops, and a variety of other needs.

About GeoPandas

GeoPandas is an open source project to add support for geographic data to pandas objects. It currently implements `GeoSeries` and `GeoDataFrame` types which are subclasses of `pandas.Series` and `pandas.DataFrame` respectively. GeoPandas objects can act on `shapely` geometry objects and perform geometric operations.

GeoPandas is a community-led project written, used and supported by a wide range of people from all around the world of a large variety of backgrounds. Want to get involved in the community? See our [community guidelines](#).

GeoPandas will always be 100% open source software, free for all to use and released under the liberal terms of the BSD-3-Clause license.

[Team](#)

[Roadmap](#)

[Citing](#)

[Logo](#)

GeoPandas is a fiscally sponsored project of NumFOCUS, a nonprofit dedicated to supporting the open-source scientific computing community. If you like GeoPandas and want to support our mission, please consider making a [donation](#) to support our efforts.

NumFOCUS is a 501(c)(3) non-profit charity in the United States; as such, donations to NumFOCUS are tax-deductible as allowed by law. As with any donation, you should consult with your personal tax adviser or the IRS about your particular tax situation.



Project history

Kelsey Jordahl founded GeoPandas project in 2013 during the Scipy Conference and released a version 0.1.0 in July 2014. In 2016, Joris Van den Bossche took the lead and became the maintainer of the project.

Since the beginning, GeoPandas is a BSD-licensed open-source project supported by a [community of contributors](#) from around the world and is now maintained by a [team](#) of core developers.

In 2020 GeoPandas became [NumFOCUS Affiliated Project](#) and received two [Small Development Grants](#) to support its development. In 2023, GeoPandas became [NumFOCUS Sponsored Project](#).

Timeline

- **2013:** Beginning of the development
 - **2014:** GeoPandas 0.1.0 released
 - **2020:** GeoPandas became [NumFOCUS Affiliated Project](#)
-

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx 7.3.7](#).

Getting Started

Installation

GeoPandas is written in pure Python, but has several dependencies written in C ([GEOS](#), [GDAL](#), [PROJ](#)). Those base C libraries can sometimes be a challenge to install. Therefore, we advise you to closely follow the recommendations below to avoid installation problems.

Easy way

The best way to install GeoPandas is using `conda` and `conda-forge` channel:

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

Detailed instructions

Do you prefer `pip install` or installation from source? Or want to install a specific version? See [detailed instructions](#).

What now?

- If you don't have GeoPandas yet, check [Installation](#).
- If you have never used GeoPandas and want to get familiar with it and its core functionality quickly, see [Getting Started Tutorial](#).
- Detailed illustration of how to work with different parts of GeoPandas, make maps, manage projections, spatially merge data or geocode are part of our [User Guide](#).
- And if you are interested in the complete documentation of all classes, functions, method and attributes GeoPandas offers, [API Reference](#) is here for you.

[Installation](#)[Introduction](#)[User Guide](#)[API Reference](#)

Get in touch

Haven't found what you were looking for?

- Ask usage questions ("How do I?") on [StackOverflow](#) or [GIS StackExchange](#).
 - Get involved in [discussions on GitHub](#)
 - Report bugs, suggest features or view the source code on [GitHub](#).
 - For a quick question about a bug report or feature request, or Pull Request, head over to the [gitter channel](#).
 - For less well defined questions or ideas, or to announce other projects of interest to GeoPandas users, ... use the [mailing list](#).
-

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

Documentation

The documentation of GeoPandas consists of four parts - [User Guide](#) with explanation of the basic functionality, [Advanced Guide](#) covering topics which assume knowledge of basics, [Examples](#), and [API reference](#) detailing every class, method, function and attribute used implemented by GeoPandas.

[User guide](#)

[Advanced guide](#)

[Examples](#)

[API reference](#)

Documentation

[User guide](#)

[Data structures](#)

[Reading and writing files](#)

[Indexing and selecting data](#)

[Making maps and plots](#)

[Interactive mapping](#)

[Projections](#)

[Geometric manipulations](#)

[Set operations with overlay](#)

[Aggregation with dissolve](#)

[Merging data](#)

[Geocoding](#)

[Sampling points](#)

[How to...](#)

[Advanced guide](#)

[Missing and empty geometries](#)

[Re-projecting using GDAL with Rasterio and Fiona](#)

[Migration from PyGEOS geometry backend to Shapely 2.0](#)

[Migration from the Fiona to the Pyogrio read/write engine](#)

[API reference](#)

[GeoSeries](#)

[GeoDataFrame](#)

[Input/output](#)

[Tools](#)

[Spatial index](#)

[Testing](#)

[Changelog](#)

[Version 1.0.1](#)

[Version 1.0.0 \(June 24, 2024\)](#)

[Version 0.14.4 \(April 26, 2024\)](#)

[Version 0.14.3 \(Jan 31, 2024\)](#)

[Version 0.14.2 \(Jan 4, 2024\)](#)

[Version 0.14.1 \(Nov 11, 2023\)](#)

[Version 0.14 \(Sep 15, 2023\)](#)

[Version 0.13.2 \(Jun 6, 2023\)](#)

[Version 0.13.1 \(Jun 5, 2023\)](#)

[Version 0.13 \(May 6, 2023\)](#)

[Version 0.12.2 \(December 10, 2022\)](#)

[Version 0.12.1 \(October 29, 2022\)](#)

[Version 0.12 \(October 24, 2022\)](#)

[Version 0.11.1 \(July 24, 2022\)](#)

[Version 0.11 \(June 20, 2022\)](#)

[Version 0.10.2 \(October 16, 2021\)](#)

[Version 0.10.1 \(October 8, 2021\)](#)

[Version 0.10.0 \(October 3, 2021\)](#)

[Version 0.9.0 \(February 28, 2021\)](#)

[Version 0.8.2 \(January 25, 2021\)](#)

[Version 0.8.1 \(July 15, 2020\)](#)

[Version 0.8.0 \(June 24, 2020\)](#)

[Version 0.7.0 \(February 16, 2020\)](#)

[Version 0.6.3 \(February 6, 2020\)](#)

[Version 0.6.2 \(November 18, 2019\)](#)

[Version 0.6.1 \(October 12, 2019\)](#)

[Version 0.6.0 \(September 27, 2019\)](#)

[Version 0.5.1 \(July 11, 2019\)](#)

[Version 0.5.0 \(April 25, 2019\)](#)

[Version 0.4.1 \(March 5, 2019\)](#)

[Version 0.4.0 \(July 15, 2018\)](#)

[Version 0.3.0 \(August 29, 2017\)](#)

[Version 0.2.0](#)

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Community

GeoPandas is a community-led project written, used and supported by a wide range of people from all around the world of a large variety of backgrounds. Everyone is welcome, each small contribution, no matter if it is a fix of a typo in the documentation, bug report, an idea, or a question, is valuable. As a member of our community, you should adhere to the principles presented in the [Code of Conduct](#).

If you'd like to contribute, please read the [Contributing guide](#). It will help you to understand the way GeoPandas development works and us to review your contribution.

GeoPandas is a part of the broader Python [ecosystem](#). It depends on a range of great tools, and various packages are built on top of GeoPandas addressing specific needs in geospatial data processing, analysis and visualization.

[Contributing](#)

[Code of Conduct](#)

[Ecosystem](#)

GeoPandas 1.0.1

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 [pyogrio](#) for file access and [matplotlib](#) for plotting.

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.

[Getting started](#)[Documentation](#)[About GeoPandas](#)[Community](#)

Useful links

[Binary Installers \(PyPI\)](#) | [Source Repository \(GitHub\)](#) | [Issues & Ideas](#) | [Q&A Support](#)

[pypi v1.0.1](#) [Tests failing](#) [codecov 96%](#) [gitter join chat](#) [launch binder](#) [powered by NumFOCUS](#)

Supported by

The GeoPandas project uses an [open governance model](#) and is fiscally sponsored by [NumFOCUS](#). Consider making a [tax-deductible donation](#) to help the project pay for developer time, professional services, travel, workshops, and a variety of other needs.

Team

Contributors

GeoPandas is developed by more than [100 volunteer contributors](#).

Core developers

- Joris Van den Bossche - **lead maintainer** | [@jorisvandenbossche](#)
- Martin Fleischmann | [@martinfleis](#)
- James McBride | [@jdmcbr](#)
- Brendan Ward | [@brendan-ward](#)
- Levi Wolf | [@ljwolf](#)
- Matt Richards | [@m-richards](#)

Founder

- Kelsey Jordahl | [@kjordahl](#)

Alumni developers

- Jacob Wasserman | [@jwass](#)

Roadmap

This page provides an overview of the strategic goals for development of GeoPandas. Some of the tasks may happen sooner given the appropriate funding, other later with no specified date, and some may not happen at all if the implementation proves to be against the will of the community or face technical issues preventing their inclusion in the code base.

The current roadmap reflects longer-term vision covering enhancements that should happen in upcoming releases.

S2 geometry engine

The geometry engine used in GeoPandas is `shapely`, which serves as a Python API for `GEOS`. It means that all geometry operations in GeoPandas are planar, using (possibly) projected coordinate reference systems. Some applications focusing on the global context may find planar operations limiting as they come with troubles around anti-meridian and poles. One solution is an implementation of a spherical geometry engine, namely `S2`, that should eliminate these limitations and offer an alternative to `GEOS`.

The GeoPandas community is currently working together with the R-spatial community that has already exposed `S2` in an R counterpart of GeoPandas `sf` on Python bindings for `S2`, that should be used as a secondary geometry engine in GeoPandas.

Prepared geometries

GeoPandas is using spatial indexing for the operations that may benefit from it. Further performance gains can be achieved using prepared geometries. Preparation creates a spatial index of individual line segments of geometries, greatly enhancing the speed of spatial predicates like `intersects` or `contains`. Given that the preparation has become less computationally expensive in `shapely` 2.0, GeoPandas should expose the preparation to the user but, more importantly, use smart automatic geometry preparation under the hood.

Static plotting improvements

GeoPandas currently covers a broad range of geospatial tasks, from data exploration to advanced analysis. However, one moment may tempt the user to use different software - plotting. GeoPandas can create static maps based on `matplotlib`, but they are a bit basic at the moment. It isn't straightforward to generate a complex map in a production-quality which can go straight to an academic journal or an infographic. We

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Citing

When citing GeoPandas, you can use [Zenodo DOI](#) for each release. Below is the example of resulting BiBTeX record for GeoPandas 0.8.1 and a reference using APA 6th ed.

Kelsey Jordahl, Joris Van den Bossche, Martin Fleischmann, Jacob Wasserman, James McBride, Jeffrey Gerard, ... François Leblanc. (2020, July 15). geopandas/geopandas: v0.8.1 (Version v0.8.1). Zenodo.
<http://doi.org/10.5281/zenodo.3946761>

```
@software{kelsey_jordahl_2020_3946761,
    author      = {Kelsey Jordahl and
                  Joris Van den Bossche and
                  Martin Fleischmann and
                  Jacob Wasserman and
                  James McBride and
                  Jeffrey Gerard and
                  Jeff Tratner and
                  Matthew Perry and
                  Adrian Garcia Badaracco and
                  Carson Farmer and
                  Geir Arne Hjelle and
                  Alan D. Snow and
                  Micah Cochran and
                  Sean Gillies and
                  Lucas Culbertson and
                  Matt Bartos and
                  Nick Eubank and
                  maxalbert and
                  Aleksey Bilogur and
                  Sergio Rey and
                  Christopher Ren and
                  Dani Arribas-Bel and
                  Leah Wasser and
                  Levi John Wolf and
                  Martin Journois and
                  Joshua Wilson and
                  Adam Greenhall and
                  Chris Holdgraf and
                  Filipe and
                  François Leblanc},
    title       = {geopandas/geopandas: v0.8.1},
    month       = jul,
    year        = 2020,
    publisher   = {Zenodo},
    version     = {v0.8.1},
    doi         = {10.5281/zenodo.3946761},
    url         = {https://doi.org/10.5281/zenodo.3946761}
}
```

GeoPandas logo

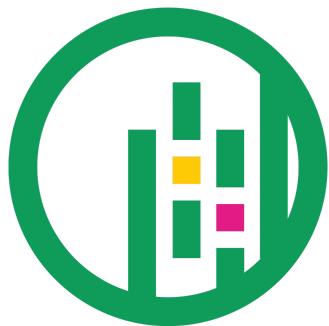
GeoPandas project uses a logo derived from [pandas logo](#), enclosing it in a globe illustrating the geographic nature of our data.

Versions

We have four versions of our logo:

Primary logo

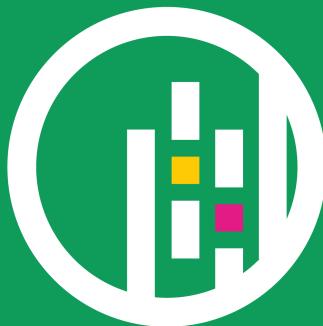
The primary logo should be used in a majority of cases. Inverted logo or icon should be used only when necessary.



GeoPandas

Inverted colors

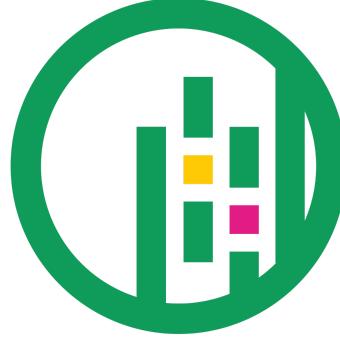
If you want to place the GeoPandas logo on a dark background, use the inverted version.



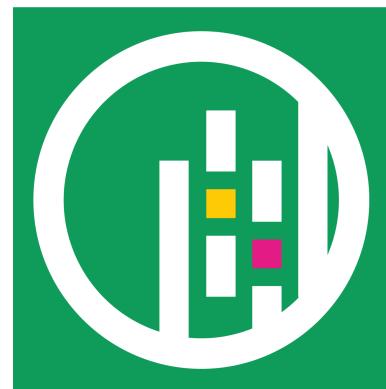
GeoPandas

Icon

Although it is possible to use icon independently, we would prefer using the complete variant above.



Inverted icon



Download

You can download all version in SVG and PNG from [GitHub repository](#).

Colors

Pink and yellow accent colors are shared with [pandas](#).

Green



HEX: #139C5A

RGB: (19, 156, 90)

Yellow



HEX: #FFCA00

RGB: (255, 202, 0)

Pink



HEX: #E70488

RGB: (231, 4, 136)

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

```
# About GeoPandas
```

```
```{toctree}
:maxdepth: 2
:caption: About
:hidden:

Team <about/team>
Roadmap <about/roadmap>
Citing <about/citing>
Logo <about/logo>
````
```

GeoPandas is an open source project to add support for geographic data to pandas objects. It currently implements `GeoSeries` and `GeoDataFrame` types which are subclasses of `pandas.Series` and `pandas.DataFrame` respectively. GeoPandas objects can act on `shapely` geometry objects and perform geometric operations.

GeoPandas is a community-led project written, used and supported by a wide range of people from all around the world of a large variety of backgrounds. Want to get involved in the community? See our [community guidelines](community).

GeoPandas will always be 100% open source software, free for all to use and released under the liberal terms of the BSD-3-Clause license.

```
```{container} button
```

```
{doc}`Team <about/team>` {doc}`Roadmap <about/roadmap>`
{doc}`Citing <about/citing>` {doc}`Logo <about/logo>`
```

GeoPandas is a fiscally sponsored project of NumFOCUS, a nonprofit dedicated to supporting the open-source scientific computing community. If you like GeoPandas and want to support our mission, please consider making a [donation](https://numfocus.org/donate-for-geopandas) to support our efforts.

NumFOCUS is a 501(c)(3) non-profit charity in the United States; as such, donations to NumFOCUS are tax-deductible as allowed by law. As with any donation, you should consult with your personal tax adviser or the IRS about your particular tax situation.

```
```{image} _static/SponsoredProject.svg
:alt: numfocus
:width: 400px
:align: center
:target: <https://numfocus.org/project/geopandas>
```

Project history

Kelsey Jordahl founded GeoPandas project in 2013 during the Scipy Conference and released a version 0.1.0 in July 2014. In 2016, Joris Van den Bossche took the lead and became the maintainer of the project. Since the beginning, GeoPandas is a BSD-licensed open-source project supported by a [community of contributors](https://github.com/geopandas/geopandas/graphs/contributors) from around the world and is now maintained by a [team](about/team) of core developers.

In 2020 GeoPandas became [NumFOCUS Affiliated Project](https://numfocus.org/sponsored-projects/affiliated-projects) and received two [Small Development Grants](https://numfocus.org/programs/sustainability) to support its development. In 2023, GeoPandas became [NumFOCUS Sponsored Project](https://numfocus.org/project/geopandas).

Timeline

- **2013**: Beginning of the development
- **2014**: GeoPandas 0.1.0 released
- **2020**: GeoPandas became [NumFOCUS Affiliated Project](https://numfocus.org/sponsored-projects/affiliated-projects)
- **2023**: GeoPandas became [NumFOCUS Sponsored Project](https://numfocus.org/project/geopandas)

Installation

GeoPandas depends for its spatial functionality on a large geospatial, open source stack of libraries ([GEOS](#), [GDAL](#), [PROJ](#)). See the [Dependencies](#) section below for more details. Those base C libraries can sometimes be a challenge to install. Therefore, we advise you to closely follow the recommendations below to avoid installation problems.

Installing with Anaconda / conda

To install GeoPandas and all its dependencies, we recommend to use the [conda](#) package manager. This can be obtained by installing the [Anaconda Distribution](#) (a free Python distribution for data science), or through [miniconda](#) (minimal distribution only containing Python and the [conda](#) package manager). See also the [Conda installation docs](#) for more information on how to install Anaconda or miniconda locally.

The advantage of using the [conda](#) package manager is that it provides pre-built binaries for all the required and optional dependencies of GeoPandas for all platforms (Windows, Mac, Linux).

To install the latest version of GeoPandas, you can then do:

```
conda install geopandas
```

Using the conda-forge channel

[conda-forge](#) is a community effort that provides conda packages for a wide range of software. It provides the *conda-forge* package channel for conda from which packages can be installed, in addition to the “*defaults*” channel provided by Anaconda. Depending on what other packages you are working with, the *defaults* channel or *conda-forge* channel may be better for your needs (e.g. some packages are available on *conda-forge* and not on *defaults*). Generally, *conda-forge* is more up to date with the latest versions of geospatial python packages.

GeoPandas and all its dependencies are available on the *conda-forge* channel, and can be installed as:

```
conda install --channel conda-forge geopandas
```

Note

We strongly recommend to either install everything from the *defaults* channel, or everything from the *conda-forge* channel. Ending up with a mixture of packages from both channels for the dependencies of GeoPandas can lead to import problems. See the [conda-forge section on using multiple channels](#) for more details.

Creating a new environment

Creating a new environment is not strictly necessary, but given that installing other geospatial packages from different channels may cause dependency conflicts (as mentioned in the note above), it can be good practice to install the geospatial stack in a clean environment starting fresh.

The following commands create a new environment with the name `geo_env`, configures it to install packages always from conda-forge, and installs GeoPandas in it:

```
conda create -n geo_env
conda activate geo_env
conda config --env --add channels conda-forge
conda config --env --set channel_priority strict
conda install python=3 geopandas
```

Installing with pip

GeoPandas can also be installed with pip, if all dependencies can be installed as well:

```
pip install geopandas
```

Warning

When using pip to install GeoPandas, you need to make sure that all dependencies are installed correctly.

Our main dependencies ([shapely](#), [pyproj](#), [pyogrio](#)) provide binary wheels with dependencies included for Mac, Linux, and Windows.

However, depending on your platform or Python version, there might be no pre-compiled wheels available, and then you need to compile and install their C dependencies manually. We refer to the individual packages for more details on installing those. Using conda (see above) avoids the need to compile the dependencies yourself.

Optional runtime dependencies can also be installed all at once:

```
pip install 'geopandas[all]'
```

Installing from source

You may install the latest development version by cloning the *GitHub* repository and using pip to install from the local directory:

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

Development dependencies can be installed using the dev optional dependency group:

```
pip install '.[dev]'
```

It is also possible to install the latest development version directly from the GitHub repository with:

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

For installing GeoPandas from source, the same [note](#) on the need to have all dependencies correctly installed applies. But, those dependencies can also be installed independently with conda before installing GeoPandas from source:

```
conda install pandas pyogrio shapely pyproj
```

See the [section on conda](#) above for more details on getting running with Anaconda.

Dependencies

Required dependencies:

- [numpy](#)
- [pandas](#) (version 1.4 or later)
- [shapely](#) (interface to [GEOS](#); version 2.0.0 or later)
- [pyogrio](#) (interface to [GDAL](#); version 0.7.2 or later)
- [pyproj](#) (interface to [PROJ](#); version 3.3.0 or later)
- [packaging](#)

Further, optional dependencies are:

- [fiona](#) (optional; slower alternative to *pyogrio*)
- [psycopg](#) (optional; for PostGIS connection)
- [psycopg2](#) (optional; for PostGIS connection - older version of *psycopg* library)
- [GeoAlchemy2](#) (optional; for writing to PostGIS)
- [geopy](#) (optional; for geocoding)
- [pointpats](#) (optional; for advanced point sampling)

For plotting, these additional packages may be used:

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Note

This page was generated from [getting_started/introduction.ipynb](#).

Interactive online version: [!\[\]\(fec5063cf6bfd35f71c9c6e0238a8491_img.jpg\) launch binder](#)

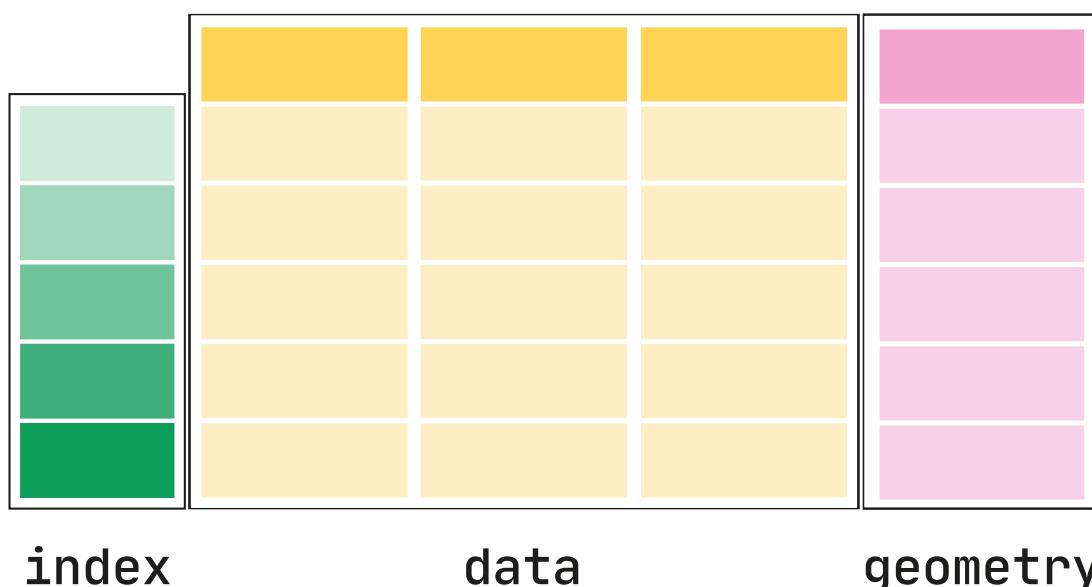
Introduction to GeoPandas

This quick tutorial introduces the key concepts and basic features of GeoPandas to help you get started with your projects.

Concepts

GeoPandas, as the name suggests, extends the popular data science library [pandas](#) by adding support for geospatial data. If you are not familiar with [pandas](#), we recommend taking a quick look at its [Getting started documentation](#) before proceeding.

The core data structure in GeoPandas is the [geopandas.GeoDataFrame](#), a subclass of [pandas.DataFrame](#), that can store geometry columns and perform spatial operations. The [geopandas.GeoSeries](#), a subclass of [pandas.Series](#), handles the geometries. Therefore, your [GeoDataFrame](#) is a combination of [pandas.Series](#), with traditional data (numerical, boolean, text etc.), and [geopandas.GeoSeries](#), with geometries (points, polygons etc.). You can have as many columns with geometries as you wish; unlike in some typical desktop GIS software.



Each [GeoSeries](#) can contain any geometry type (you can even mix them within a single array) and has a [GeoSeries.crs](#) attribute, which stores information about the projection (CRS stands for Coordinate Reference System). Therefore, each [GeoSeries](#) in a [GeoDataFrame](#) can be in a different projection, allowing you to have, for example, multiple versions (different projections) of the same geometry.

Only one `GeoSeries` in a `GeoDataFrame` is considered the *active geometry column*, which means that all geometric operations applied to a `GeoDataFrame` operate on this column. The active geometry column is accessed via the `GeoDataFrame.geometry` attribute.

i User guide

See more on [data structures](#) in the user guide.

Let's see how some of these concepts work in practice.

Reading and writing files

First, we need to read some data.

Reading files

Assuming you have a file containing both data and geometry (e.g. GeoPackage, GeoJSON, Shapefile), you can read it using `geopandas.read_file()`, which automatically detects the filetype and creates a `GeoDataFrame`. This tutorial uses the `"nybb"` dataset, a map of New York boroughs, which is available through the `geodatasets` package. Therefore, we use `geodatasets.get_path()` to download the dataset and retrieve the path to the local copy.

```
[1]: import geopandas  
from geodatasets import get_path  
  
path_to_data = get_path("nybb")  
gdf = geopandas.read_file(path_to_data)  
  
gdf
```

	BoroCode	BoroName	Shape_Leng	Shape_Area	geometry
0	5	Staten Island	330470.010332	1.623820e+09	MULTIPOLYGON (((970217.022 145643.332, 970227....
1	4	Queens	896344.047763	3.045213e+09	MULTIPOLYGON (((1029606.077 156073.814, 102957....
2	3	Brooklyn	741080.523166	1.937479e+09	MULTIPOLYGON (((1021176.479 151374.797, 102100....
3	1	Manhattan	359299.096471	6.364715e+08	MULTIPOLYGON (((981219.056 188655.316, 980940....
4	2	Bronx	464392.991824	1.186925e+09	MULTIPOLYGON (((1012821.806 229228.265, 101278....

Writing files

To write a `GeoDataFrame` back to file use `GeoDataFrame.to_file()`. By default, the file format is inferred by the file extension (e.g. `.shp` for Shapefile, `.geojson` for GeoJSON), but you can specify this explicitly with the `driver` keyword.

```
[2]: gdf.to_file("my_file.geojson", driver="GeoJSON")
```

i User guide

See more on [reading and writing files](#) in the user guide.

Simple accessors and methods

Now we have our `GeoDataFrame` and can start working with its geometry.

Since there was only one geometry column in the New York Boroughs dataset, this column automatically becomes the *active* geometry and spatial methods used on the `GeoDataFrame` will be applied to the `"geometry"` column.

Measuring area

To measure the area of each polygon (or MultiPolygon in this specific case), access the `GeoDataFrame.area` attribute, which returns a `pandas.Series`. Note that `GeoDataFrame.area` is just `GeoSeries.area` applied to the *active* geometry column.

But first, to make the results easier to read, set the names of the boroughs as the index:

```
[3]: gdf = gdf.set_index("BoroName")
```

```
[4]: gdf["area"] = gdf.area  
gdf["area"]
```

```
[4]: BoroName  
Staten Island      1.623822e+09  
Queens            3.045214e+09  
Brooklyn          1.937478e+09  
Manhattan         6.364712e+08  
Bronx             1.186926e+09  
Name: area, dtype: float64
```

Getting polygon boundary and centroid

To get the boundary of each polygon (LineString), access the `GeoDataFrame.boundary`:

```
[5]: gdf["boundary"] = gdf.boundary  
gdf["boundary"]
```

```
[5]: BoroName
Staten Island      MULTILINESTRING ((970217.022 145643.332, 97022...
Queens            MULTILINESTRING ((1029606.077 156073.814, 1029...
Brooklyn          MULTILINESTRING ((1021176.479 151374.797, 1021...
Manhattan         MULTILINESTRING ((981219.056 188655.316, 98094...
Bronx             MULTILINESTRING ((1012821.806 229228.265, 1012...
Name: boundary, dtype: geometry
```

Since we have saved boundary as a new column, we now have two geometry columns in the same `GeoDataFrame`.

We can also create new geometries, which could be, for example, a buffered version of the original one (i.e., `GeoDataFrame.buffer(10)`) or its centroid:

```
[6]: gdf["centroid"] = gdf.centroid
gdf["centroid"]
```

```
[6]: BoroName
Staten Island      POINT (941639.45 150931.991)
Queens            POINT (1034578.078 197116.604)
Brooklyn          POINT (998769.115 174169.761)
Manhattan         POINT (993336.965 222451.437)
Bronx             POINT (1021174.79 249937.98)
Name: centroid, dtype: geometry
```

Measuring distance

We can also measure how far each centroid is from the first centroid location.

```
[7]: first_point = gdf["centroid"].iloc[0]
gdf["distance"] = gdf["centroid"].distance(first_point)
gdf["distance"]
```

```
[7]: BoroName
Staten Island      0.000000
Queens            103781.535276
Brooklyn          61674.893421
Manhattan         88247.742789
Bronx             126996.283623
Name: distance, dtype: float64
```

Note that `geopandas.GeoDataFrame` is a subclass of `pandas.DataFrame`, so we have all the pandas functionality available to use on the geospatial dataset — we can even perform data manipulations with the attributes and geometry information together.

For example, to calculate the average of the distances measured above, access the ‘distance’ column and call the `mean()` method on it:

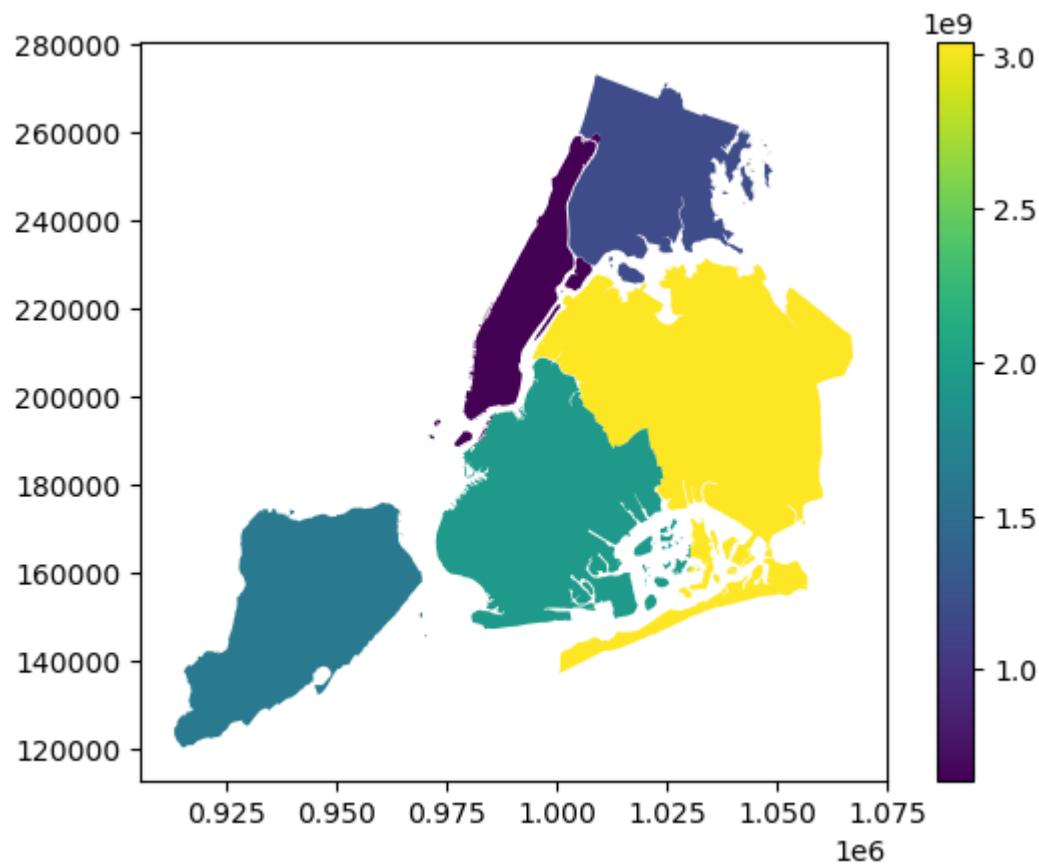
```
[8]: gdf["distance"].mean()
[8]: np.float64(76140.09102166798)
```

Making maps

GeoPandas can also plot maps, so we can check how the geometries appear in space. To plot the active geometry, call `GeoDataFrame.plot()`. To color code by another column, pass in that column as the first argument. In the example below, we plot the active geometry column and color code by the `"area"` column. We also want to show a legend (`legend=True`).

```
[9]: gdf.plot("area", legend=True)
```

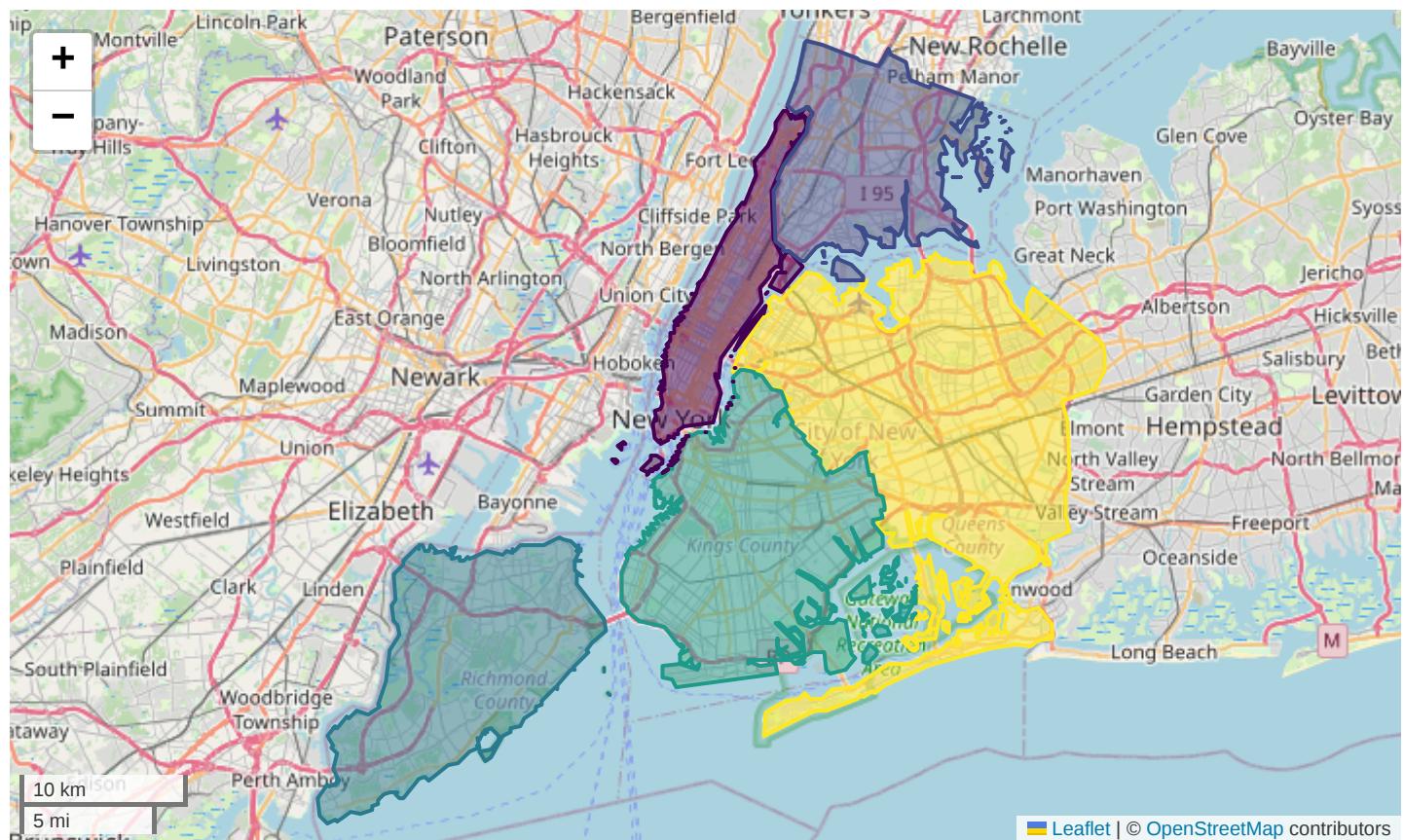
```
[9]: <Axes: >
```



You can also explore your data interactively using `GeoDataFrame.explore()`, which behaves in the same way `plot()` does but returns an interactive map instead.

```
[10]: gdf.explore("area", legend=False)
```

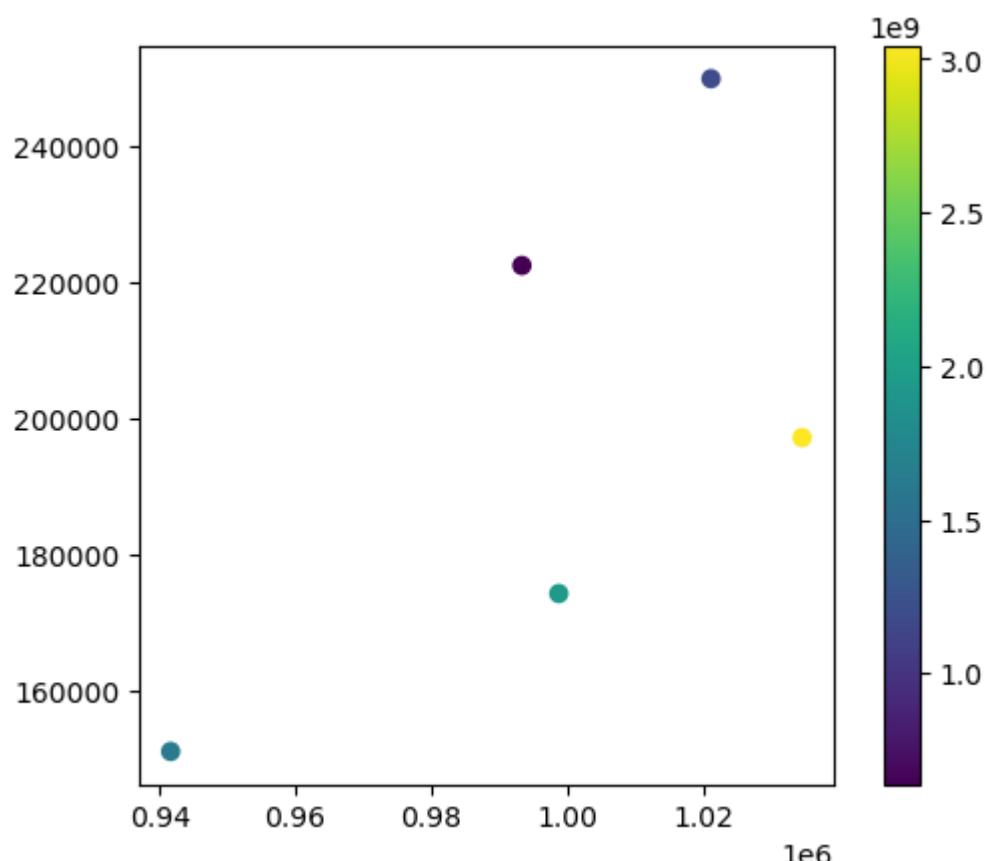
[10]:



Switching the active geometry (`GeoDataFrame.set_geometry`) to centroids, we can plot the same data using point geometry.

```
[11]: gdf = gdf.set_geometry("centroid")
gdf.plot("area", legend=True)
```

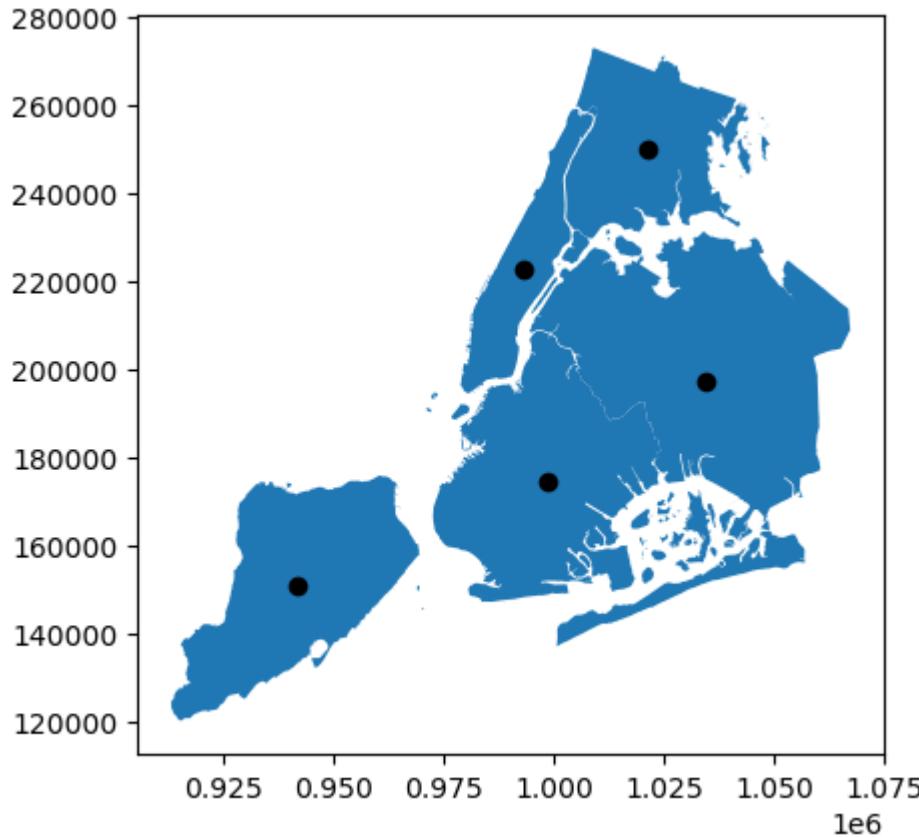
```
[11]: <Axes: >
```



And we can also layer both `GeoSeries` on top of each other. We just need to use one plot as an axis for the other.

```
[12]: ax = gdf["geometry"].plot()  
gdf["centroid"].plot(ax=ax, color="black")
```

[12]: <Axes: >



Now we set the active geometry back to the original `GeoSeries`.

```
[13]: gdf = gdf.set_geometry("geometry")
```

i User guide

See more on [mapping](#) in the user guide.

Geometry creation

We can further work with the geometry and create new shapes based on those we already have.

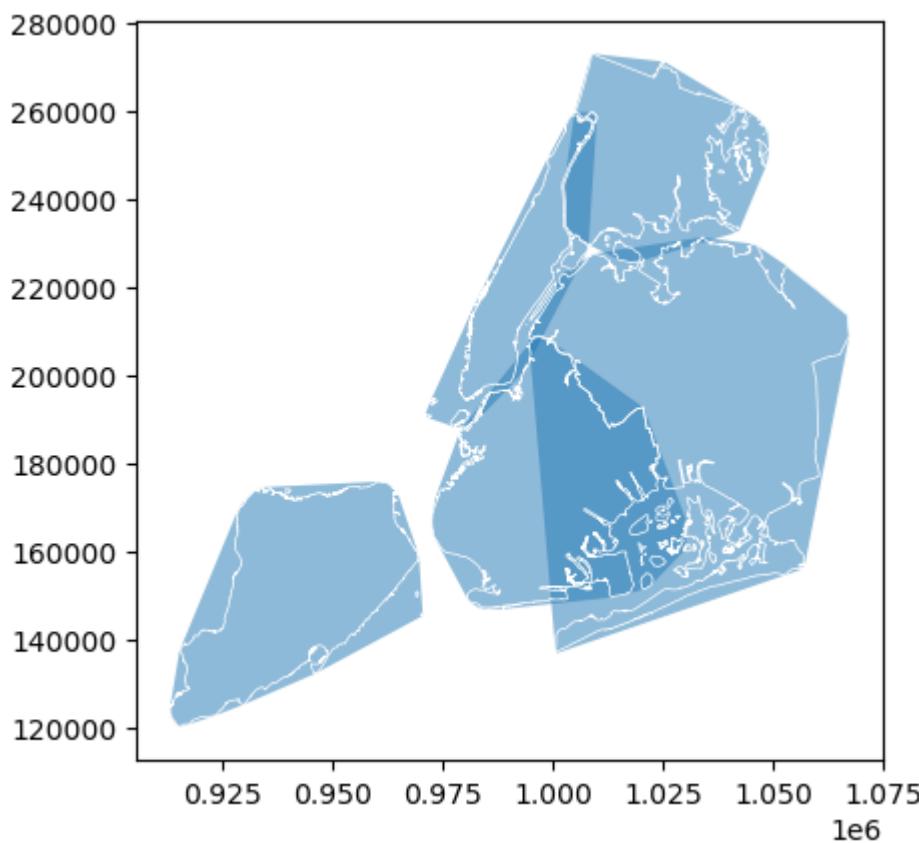
Convex hull

If we are interested in the convex hull of our polygons, we can access `GeoDataFrame.convex_hull`.

```
[14]: gdf["convex_hull"] = gdf.convex_hull
```

```
[15]: # saving the first plot as an axis and setting alpha (transparency) to 0.5  
ax = gdf["convex_hull"].plot(alpha=0.5)  
# passing the first plot and setting linewidth to 0.5  
gdf["boundary"].plot(ax=ax, color="white", linewidth=0.5)
```

[15]: <Axes: >



Buffer

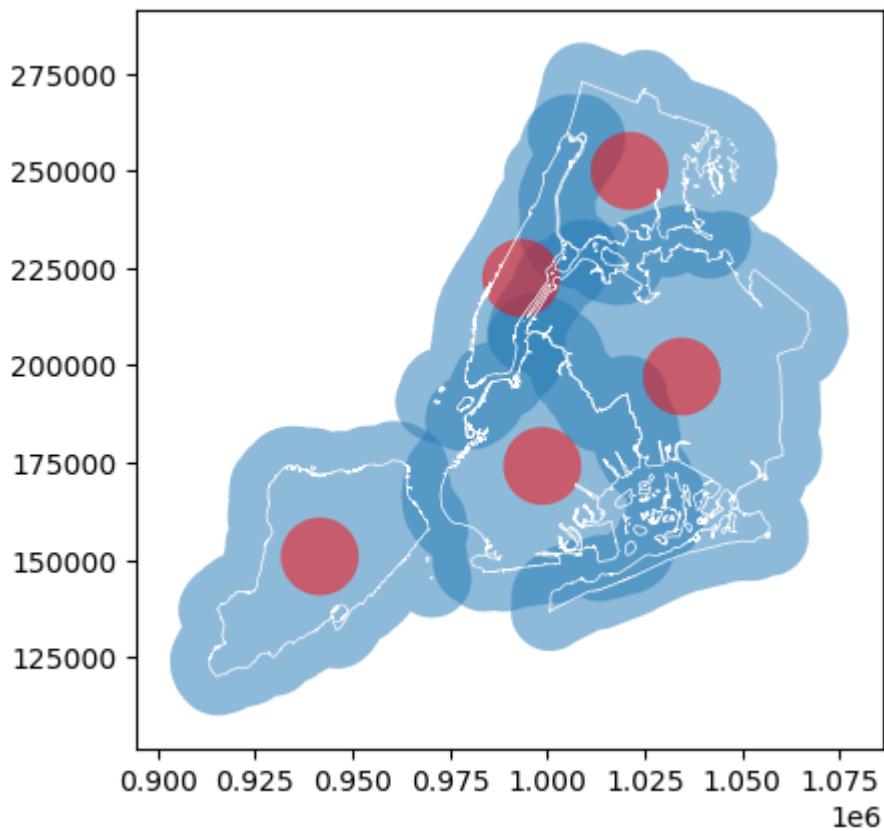
In other cases, we may need to buffer the geometry using `GeoDataFrame.buffer()`. Geometry methods are automatically applied to the active geometry, but we can apply them directly to any `GeoSeries` as well. Let's buffer the boroughs and their centroids and plot both on top of each other.

```
[16]: # buffering the active geometry by 10 000 feet (geometry is already in feet)
gdf["buffered"] = gdf.buffer(10000)

# buffering the centroid geometry by 10 000 feet (geometry is already in feet)
gdf["buffered_centroid"] = gdf["centroid"].buffer(10000)
```

```
[17]: # saving the first plot as an axis and setting alpha (transparency) to 0.5
ax = gdf["buffered"].plot(alpha=0.5)
# passing the first plot as an axis to the second
gdf["buffered_centroid"].plot(ax=ax, color="red", alpha=0.5)
# passing the first plot and setting linewidth to 0.5
gdf["boundary"].plot(ax=ax, color="white", linewidth=0.5)
```

[17]: <Axes: >



i User guide

See more on [geometry manipulations](#) in the user guide.

Geometry relations

We can also ask about the spatial relations of different geometries. Using the geometries above, we can check which of the buffered boroughs intersect the original geometry of Brooklyn, i.e., is within 10 000 feet from Brooklyn.

First, we get a polygon of Brooklyn.

```
[18]: brooklyn = gdf.loc["Brooklyn", "geometry"]
brooklyn
```

[18]:



The polygon is a [shapely geometry object](#), as any other geometry used in GeoPandas.

[19]: `type(brooklyn)`

[19]: `shapely.geometry.multipolygon.MultiPolygon`

Then we can check which of the geometries in `gdf["buffered"]` intersects it.

[20]: `gdf["buffered"].intersects(brooklyn)`

[20]:

BoroName	
Staten Island	True
Queens	True
Brooklyn	True
Manhattan	True
Bronx	False

`dtype: bool`

Only Bronx (on the north) is more than 10 000 feet away from Brooklyn. All the others are closer and intersect our polygon.

Alternatively, we can check which buffered centroids are entirely within the original boroughs polygons. In this case, both `GeoSeries` are aligned, and the check is performed for each row.

[21]: `gdf["within"] = gdf["buffered_centroid"].within(gdf)`
`gdf["within"]`

[21]:

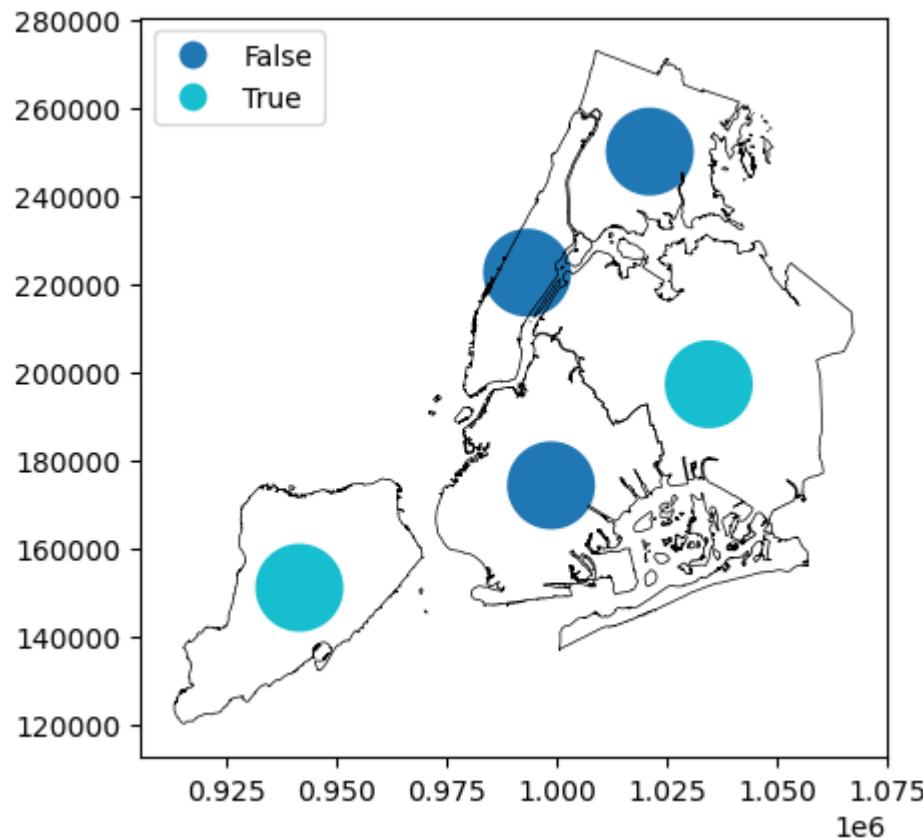
BoroName	
Staten Island	True
Queens	True
Brooklyn	False
Manhattan	False
Bronx	False

`Name: within, dtype: bool`

We can plot the results on the map to confirm the finding.

```
[22]: gdf = gdf.set_geometry("buffered_centroid")
# using categorical plot and setting the position of the legend
ax = gdf.plot(
    "within", legend=True, categorical=True, legend_kwds={"loc": "upper left"}
)
# passing the first plot and setting linewidth to 0.5
gdf["boundary"].plot(ax=ax, color="black", linewidth=0.5)
```

[22]: <Axes: >



Projections

Each `GeoSeries` has its Coordinate Reference System (CRS) accessible at `GeoSeries.crs`. The CRS tells GeoPandas where the coordinates of the geometries are located on the earth's surface. In some cases, the CRS is geographic, which means that the coordinates are in latitude and longitude. In those cases, its CRS is WGS84, with the authority code `EPSG:4326`. Let's see the projection of our NY boroughs `GeoDataFrame`.

```
[23]: gdf.crs
```

```
[23]: <Projected CRS: EPSG:2263>
  Name: NAD83 / New York Long Island (ftUS)
  Axis Info [cartesian]:
    - X[east]: Easting (US survey foot)
    - Y[north]: Northing (US survey foot)
  Area of Use:
    - name: United States (USA) - New York - counties of Bronx; Kings; Nassau; New York; C
    - bounds: (-74.26, 40.47, -71.8, 41.3)
  Coordinate Operation:
    - name: SPCS83 New York Long Island zone (US Survey feet)
    - method: Lambert Conic Conformal (2SP)
  Datum: North American Datum 1983
  - Ellipsoid: GRS 1980
  - Prime Meridian: Greenwich
```

Geometries are in `EPSG:2263` with coordinates in feet. We can easily re-project a `GeoSeries` to another CRS, like `EPSG:4326` using `GeoSeries.to_crs()`.

```
[24]: gdf = gdf.set_geometry("geometry")
boroughs_4326 = gdf.to_crs("EPSG:4326")
boroughs_4326.plot()
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

geopandas.org

Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

geopandas.org

Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

geopandas.org

Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

API reference

The API reference provides an overview of all public objects, functions and methods implemented in GeoPandas. All classes and function exposed in `geopandas.*` namespace plus those listed in the reference are public.

⚠ Warning

The `geopandas.array` and `geopandas.base` modules are private. Stable functionality in such modules is not guaranteed.

[GeoSeries](#)

[Constructor](#)

[General methods and attributes](#)

[Unary predicates](#)

[Binary predicates](#)

[Set-theoretic methods](#)

[Constructive methods and attributes](#)

[Affine transformations](#)

[Linestring operations](#)

[Aggregating and exploding](#)

[Serialization / IO / conversion](#)

[Projection handling](#)

[Missing values](#)

[Overlay operations](#)

[Plotting](#)

[Spatial index](#)

[Indexing](#)

[Interface](#)

[GeoDataFrame](#)

[Constructor](#)

[Serialization / IO / conversion](#)

[Projection handling](#)

[Active geometry handling](#)

[Aggregating and exploding](#)

[Spatial joins](#)

[Overlay operations](#)

[Plotting](#)

[Spatial index](#)

[Indexing](#)

[Interface](#)

[Input/output](#)

[GIS vector files](#)

[PostGIS](#)

[Feather](#)

[Parquet](#)

[Tools](#)

[geopandas.sjoin](#)

[geopandas.sjoin_nearest](#)

[geopandas.overlay](#)

[geopandas.clip](#)

[geopandas.tools.geocode](#)

[geopandas.tools.reverse_geocode](#)

[geopandas.tools.collect](#)

[geopandas.points_from_xy](#)

[Spatial index](#)

[Constructor](#)

[Spatial index object](#)

[Testing](#)

[geopandas.testing.assert_geoseries_equal](#)

[geopandas.testing.assert_geodataframe_equal](#)

```
# Getting Started

```{toctree}

maxdepth: 2
caption: Getting Started
hidden:

Installation <getting_started/install>
Introduction to GeoPandas <getting_started/introduction>
Examples Gallery <gallery/index>
```

## Installation

GeoPandas is written in pure Python, but has several dependencies written in C ([GEOS](https://geos.osgeo.org), [GDAL](https://www.gdal.org/), [PROJ](https://proj.org/)). Those base C libraries can sometimes be a challenge to install. Therefore, we advise you to closely follow the recommendations below to avoid installation problems.

### Easy way

The best way to install GeoPandas is using ``conda`` and ``conda-forge`` channel:

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

### Detailed instructions

Do you prefer ``pip install`` or installation from source? Or want to install a specific version? See {doc}`detailed instructions <getting_started/install>`.

### What now?



- If you don't have GeoPandas yet, check {doc}`Installation <getting_started/install>`.
- If you have never used GeoPandas and want to get familiar with it and its core functionality quickly, see {doc}`Getting Started Tutorial <getting_started/introduction>`.
- Detailed illustration of how to work with different parts of GeoPandas, make maps, manage projections, spatially merge data or geocode are part of our {doc}`User Guide <docs/user_guide>`.
- And if you are interested in the complete documentation of all classes, functions, method and attributes GeoPandas offers, {doc}`API Reference <docs/reference>` is here for you.



```{container} button

{doc}`Installation <getting_started/install>` {doc}`Introduction <getting_started/introduction>` {doc}`User Guide <docs/user_guide>` {doc}`API Reference <docs/reference>`
```

# Data structures

GeoPandas implements two main data structures, a [GeoSeries](#) and a [GeoDataFrame](#). These are subclasses of [pandas.Series](#) and [pandas.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](#) do not need to 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 [GeoSeries API reference](#). 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 the [Simple Feature Access](#) standard.

## Basic methods

- `distance()`: 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

- `geom_equals_exact()`: is shape the same as `other` (up to a specified decimal place tolerance)
- `contains()`: is shape contained within `other`
- `intersects()`: 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 - the “active geometry column”. When a spatial method is applied to a [GeoDataFrame](#) (or a spatial attribute like `area` is called), these operations will always act on the active geometry column.

The active geometry column – no matter the name of the corresponding [GeoSeries](#) – can be accessed through the `geometry` attribute (`gdf.geometry`), and the name of the `geometry` column can be found by typing `gdf.geometry.name` or `gdf.active_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 [`GeoDataFrame.set\_geometry\(\)`](#) method.

An example using the `geoda.malaria` dataset from [geodatasets](#) containing the counties of Colombia:

&gt;&gt;&gt;

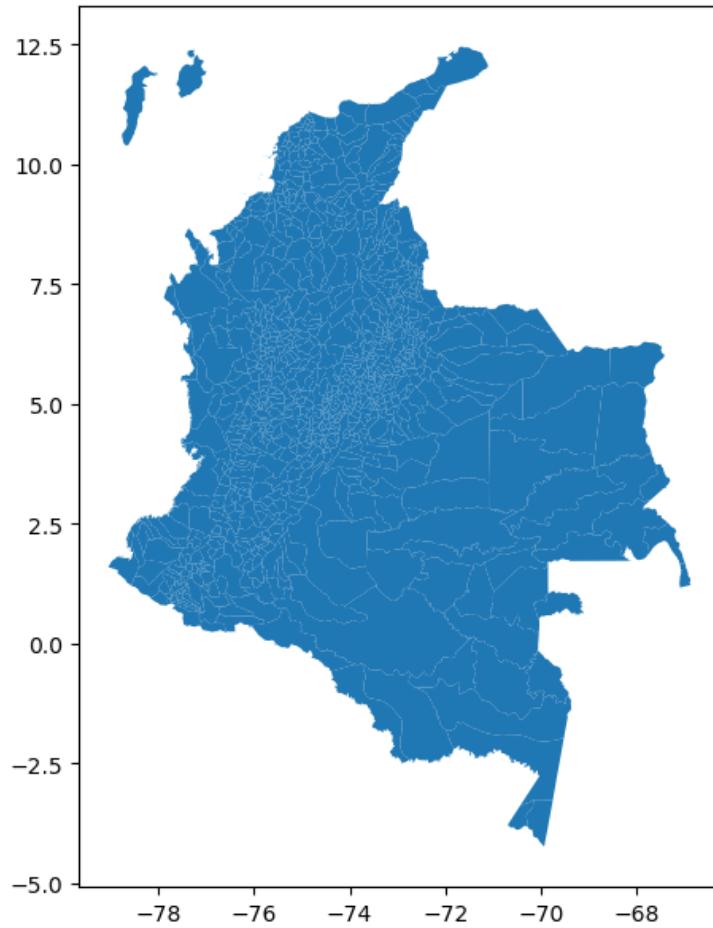
In [1]: `import geodatasets`In [2]: `colombia = geopandas.read_file(geodatasets.get_path('geoda.malaria'))`In [3]: `colombia.head()`

Out[3]:

ID	ADMO	...	RP2005	geometry
0	1	COLOMBIA	...	61773 POLYGON ((-71.32639 11.84789, -71.33579 11.855...
1	2	COLOMBIA	...	36465 POLYGON ((-72.42191 11.79824, -72.4198 11.795, ...
2	3	COLOMBIA	...	18368 POLYGON ((-72.1891 11.5242, -72.1833 11.5323, ...
3	4	COLOMBIA	...	7566 POLYGON ((-72.638 11.3679, -72.6259 11.3499, -...
4	5	COLOMBIA	...	9343 POLYGON ((-74.77489 10.93158, -74.7753 10.9338...

[5 rows x 51 columns]

# Plot countries

In [4]: `colombia.plot(markersize=.5);`

Currently, the column named “geometry” with county borders is the active geometry column:

&gt;&gt;&gt;

In [5]: `colombia.geometry.name`

Out[5]: 'geometry'

You can also rename this column to “borders”:

&gt;&gt;&gt;

```
In [6]: colombia = colombia.rename_geometry('borders')
```

```
In [7]: colombia.geometry.name
```

```
Out[7]: 'borders'
```

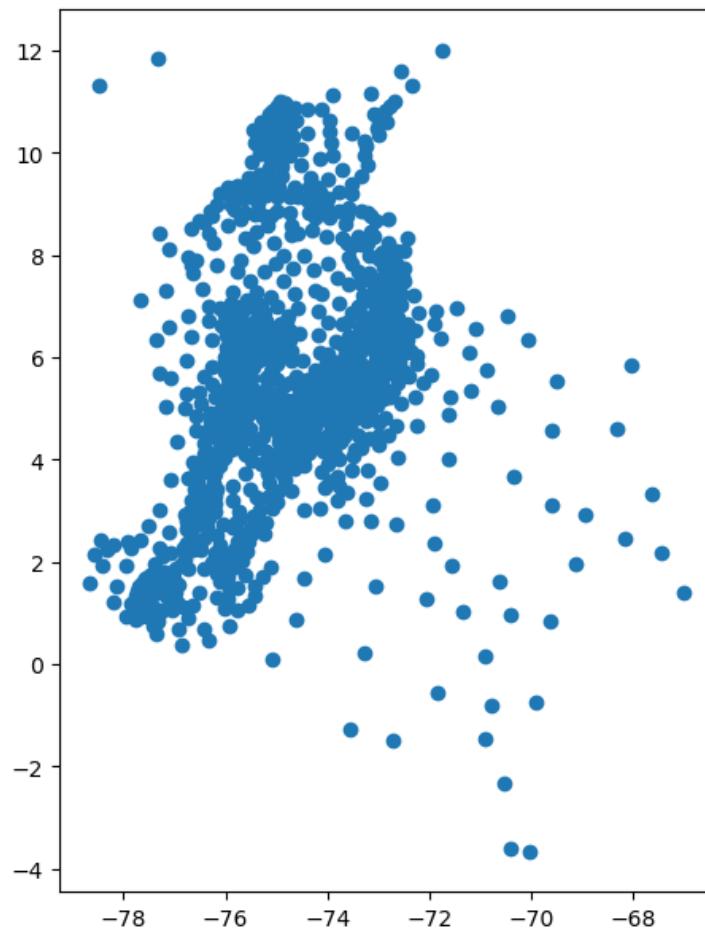
Now, you create centroids and make it the geometry:

&gt;&gt;&gt;

```
In [8]: colombia['centroid_column'] = colombia.centroid
```

```
In [9]: colombia = colombia.set_geometry('centroid_column')
```

```
In [10]: colombia.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` – they are just applied to the active geometry column `GeoSeries`.

However, `GeoDataFrames` also have a number few extra methods for:

- [Reading and writing files](#)
- [Spatial joins](#)
- [Spatial aggregations](#)
- [Geocoding](#)

## Display options

GeoPandas has an `options` attribute with global configuration attributes:

```
In [11]: import geopandas >>>
In [12]: geopandas.options
Out[12]:
Options(
 display_precision: None [default: None]
 The precision (maximum number of decimals) of the coordinates in the
 WKT representation in the Series/DataFrame display. By default (None),
 it tries to infer and use 3 decimals for projected coordinates and 5
 decimals for geographic coordinates.
 use_pygeos: False [default: False]
 Deprecated option previously used to enable PyGEOS. It will be removed
 in GeoPandas 1.1.
 io_engine: None [default: None]
 The default engine for ``read_file`` and ``to_file``. Options are
 'pyogrio' and 'fiona'.
)
```

The `geopandas.options.display_precision` option can control the number of decimals to show in the display of coordinates in the geometry column. In the `colombia` example of above, the default is to show 5 decimals for geographic coordinates:

```
In [13]: colombia['centroid_column'].head()
Out[13]:
0 POINT (-71.74594 12.00885)
1 POINT (-72.56514 11.58174)
2 POINT (-72.35203 11.32204)
3 POINT (-73.14121 11.15251)
4 POINT (-74.64555 10.88454)
Name: centroid_column, dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

# 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 `geopandas.read_file()` command:

```
geopandas.read_file(...)
```

which returns a GeoDataFrame object. This is possible because GeoPandas makes use of the massive open-source program called [GDAL/OGR](#) designed to facilitate spatial data transformations, through the Python packages [Pyogrio](#) or [Fiona](#), which both provide bindings to GDAL.

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

```
import pyogrio; help(pyogrio.read_dataframe)
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:

```
countries_gdf = geopandas.read_file("package.gpkg", layer='countries')
```

If you have a file with multiple layers, you can list them using `geopandas.list_layers()`. Note that this function requires Pyogrio.

GeoPandas can also load resources directly from a web URL, for example for GeoJSON files from [geojson.xyz](#):

```
url = "http://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_110m_land.geojson"
df = geopandas.read_file(url)
```

You can also load ZIP files that contain your data:

```
zipfile = "zip:///Users/name/Downloads/cb_2017_us_state_500k.zip"
states = geopandas.read_file(zipfile)
```

If the dataset is in a folder in the ZIP file, you have to append its name:

```
zipfile = "zip:///Users/name/Downloads/gadm36_AFG_shp.zip!data"
```

If there are multiple datasets in a folder in the ZIP file, you also have to specify the filename:

```
zipfile = "zip:///Users/name/Downloads/gadm36_AFG_shp.zip!data/gadm36_AFG_1.shp"
```

It is also possible to read any file-like objects with a [read\(\)](#) method, such as a file handler (e.g. via built-in [open\(\)](#) function) or [StringIO](#):

```
filename = "test.geojson"
file = open(filename)
df = geopandas.read_file(file)
```

File-like objects from [fsspec](#) can also be used to read data, allowing for any combination of storage backends and caching supported by that project:

```
path = "simplecache::http://download.geofabrik.de/antarctica-latest-free.shp.zip"
with fsspec.open(path) as file:
 df = geopandas.read_file(file)
```

You can also read path objects:

```
import pathlib
path_object = pathlib.Path(filename)
df = geopandas.read_file(path_object)
```

## Reading subsets of the data

Since geopandas is powered by GDAL, you can take advantage of pre-filtering when loading in larger datasets. This can be done geospatially with a geometry or bounding box. You can also filter rows loaded with a slice. Read more at [geopandas.read\\_file\(\)](#).

## Geometry filter

The geometry filter only loads data that intersects with the geometry.

```
import geodatasets

gdf_mask = geopandas.read_file(
 geodatasets.get_path("geoda.nyc")
)
gdf = geopandas.read_file(
 geodatasets.get_path("geoda.nyc education"),
 mask=gdf_mask[gdf_mask.name=="Coney Island"],
)
```

## Bounding box filter

The bounding box filter only loads data that intersects with the bounding box.

```
bbox = (
 1031051.7879884212, 224272.49231459625, 1047224.3104931959, 244317.30894023244
)
gdf = geopandas.read_file(
 geodatasets.get_path("nybb"),
 bbox=bbox,
)
```

## Row filter

Filter the rows loaded in from the file using an integer (for the first n rows) or a slice object.

```
gdf = geopandas.read_file(
 geodatasets.get_path("geoda.nyc"),
 rows=10,
)
gdf = geopandas.read_file(
 geodatasets.get_path("geoda.nyc"),
 rows=slice(10, 20),
)
```

## Field/column filters

Load in a subset of fields from the file using the `columns` keyword (this requires pyogrio or Fiona 1.9+):

```
gdf = geopandas.read_file(
 geodatasets.get_path("geoda.nyc"),
 columns=["name", "rent2008", "kids2000"],
)
```

Skip loading geometry from the file:

## i Note

Returns [pandas.DataFrame](#)

```
pdf = geopandas.read_file(
 geodatasets.get_path("geoda.nyc"),
 ignore_geometry=True,
)
```

## SQL WHERE filter

! **Added in version 0.12.**

Load in a subset of data with a [SQL WHERE clause](#).

## i Note

Requires Fiona 1.9+ or the pyogrio engine.

```
gdf = geopandas.read_file(
 geodatasets.get_path("geoda.nyc"),
 where="subborough='Coney Island'",
)
```

## Supported drivers / file formats

When using pyogrio, all drivers supported by the GDAL installation are enabled, and you can check those with:

```
import pyogrio; pyogrio.list_drivers()
```

where the values indicate whether reading, writing or both are supported for a given driver. Fiona only exposes a default subset of drivers. To display those, type:

```
import fiona; fiona.supported_drivers
```

There is a [list of available drivers](#) which are unexposed by default but may be supported (depending on the GDAL-build). You can activate these at runtime by updating the `supported_drivers` dictionary like:

```
fiona.supported_drivers["NAS"] = "raw"
```

# Writing spatial data

GeoDataFrames can be exported to many different standard formats using the

```
geopandas.GeoDataFrame.to_file\(\) method. For a full list of supported formats, type import pyogrio;
pyogrio.list_drivers\(\).
```

In addition, GeoDataFrames can be uploaded to [PostGIS](#) database (starting with GeoPandas 0.8) by using the [geopandas.GeoDataFrame.to\\_postgis\(\)](#) method.

## Note

GeoDataFrame can contain more field types than supported by most of the file formats. For example tuples or lists can be easily stored in the GeoDataFrame, but saving them to e.g. GeoPackage or Shapefile will raise a ValueError. Before saving to a file, they need to be converted to a format supported by a selected driver.

## Note

One GeoDataFrame can contain multiple geometry (GeoSeries) columns, but most standard GIS file formats, e.g. GeoPackage or ESRI Shapefile, support only a single geometry column. To store multiple geometry columns, non-active GeoSeries need to be converted to an alternative representation like well-known text (WKT) or well-known binary (WKB) before saving to file. Alternatively, they can be saved as an Apache (Geo)Parquet or Feather file, both of which support multiple geometry columns natively.

## Writing to Shapefile:

```
countries_gdf.to_file("countries.shp")
```

## Writing to GeoJSON:

```
countries_gdf.to_file("countries.geojson", driver='GeoJSON')
```

## Writing to GeoPackage:

```
countries_gdf.to_file("package.gpkg", layer='countries', driver="GPKG")
cities_gdf.to_file("package.gpkg", layer='cities', driver="GPKG")
```

## Writing with multiple geometry columns:

```
countries_gdf["country_center"] = countries_gdf["geometry"].centroid
Line below fails because GeoJSON can't contain multiple geometry columns
countries_gdf.to_file("countries.geojson", driver='GeoJSON')
countries_gdf["country_center"] = countries_gdf["country_center"].to_wkt()
countries_gdf.to_file("countries.geojson", driver='GeoJSON')
```

For multi-layer formats such as GeoPackage, it is possible to write additional geometry columns to separate layers instead of saving them as WKT or WKB within a single layer.

## Spatial databases

GeoPandas can also get data from a PostGIS database using the [`geopandas.read\_postgis\(\)`](#) command.

Writing to PostGIS:

```
from sqlalchemy import create_engine
db_connection_url = "postgresql://myusername:mypassword@myhost:5432/mydatabase";
engine = create_engine(db_connection_url)
countries_gdf.to_postgis("countries_table", con=engine)
```

## Apache Parquet and Feather file formats

**!** *Added in version 0.8.0.*

GeoPandas supports writing and reading the Apache Parquet and Feather file formats.

[Apache Parquet](#) is an efficient, columnar storage format (originating from the Hadoop ecosystem). It is a widely used binary file format for tabular data. The Feather file format is the on-disk representation of the [Apache Arrow](#) memory format, an open standard for in-memory columnar data.

The [`geopandas.read\_parquet\(\)`](#), [`geopandas.read\_feather\(\)`](#), [`GeoDataFrame.to\_parquet\(\)`](#) and [`GeoDataFrame.to\_feather\(\)`](#) methods enable fast roundtrip from GeoPandas to those binary file formats, preserving the spatial information.

# Indexing and selecting data

GeoPandas inherits the standard [pandas](#) methods for indexing/selecting data. This includes label based indexing with [loc](#) and integer position based indexing with [iloc](#), which apply to both [GeoSeries](#) and [GeoDataFrame](#) objects. For more information on indexing/selecting, see the [pandas](#) documentation.

In addition to the standard [pandas](#) methods, GeoPandas also provides coordinate based indexing with the [cx](#) indexer, which slices using a bounding box. Geometries in the [GeoSeries](#) or [GeoDataFrame](#) that intersect the bounding box will be returned.

Using the [geoda.chile\\_labor](#) dataset, you can use this functionality to quickly select parts of Chile whose boundaries extend south of the -50 degrees latitude. You can first check the original GeoDataFrame.

```
In [1]: import geodatasets >>>
In [2]: chile = geopandas.read_file(geodatasets.get_path('geoda.chile_labor'))
In [3]: chile.plot(figsize=(8, 8));
```



© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# Merging data

There are two ways to combine datasets in GeoPandas – attribute joins and spatial joins.

In an attribute join, a `GeoSeries` or `GeoDataFrame` is combined with a regular `pandas.Series` or `pandas.DataFrame` based on a common variable. This is analogous to normal merging or joining in pandas.

In a spatial join, observations from two `GeoSeries` or `GeoDataFrame` are combined based on their spatial relationship to one another.

In the following examples, these datasets are used:

```
>>>
In [1]: import geodatasets
In [2]: chicago = geopandas.read_file(geodatasets.get_path("geoda.chicago_commpop"))
In [3]: groceries = geopandas.read_file(geodatasets.get_path("geoda.groceries"))

For attribute join
In [4]: chicago_shapes = chicago[['geometry', 'NID']]

In [5]: chicago_names = chicago[['community', 'NID']]

For spatial join
In [6]: chicago = chicago[['geometry', 'community']].to_crs(groceries.crs)
```

# Appending

Appending `GeoDataFrame` and `GeoSeries` uses pandas `concat()` function. Keep in mind, that appended geometry columns needs to have the same CRS.

```
>>>
Appending GeoSeries
In [7]: joined = pd.concat([chicago.geometry, groceries.geometry])

Appending GeoDataFrames
In [8]: douglas = chicago[chicago.community == 'DOUGLAS']

In [9]: oakland = chicago[chicago.community == 'OAKLAND']

In [10]: douglas_oakland = pd.concat([douglas, oakland])
```

# Attribute joins

Attribute joins are accomplished using the `merge()` method. In general, it is recommended to use the `merge()` method called from the spatial dataset. With that said, the stand-alone `pandas.merge()` function

will work if the `GeoDataFrame` is in the `left` argument; if a `DataFrame` is in the `left` argument and a `GeoDataFrame` is in the `right` position, the result will no longer be a `GeoDataFrame`.

For example, consider the following merge that adds full names to a `GeoDataFrame` that initially has only area ID for each geometry by merging it with a `DataFrame`.

```
`chicago_shapes` is GeoDataFrame with community shapes and area IDs
In [11]: chicago_shapes.head()
Out[11]:
 geometry NID
0 MULTIPOLYGON (((-87.609140876 41.844692503, -8... 35
1 MULTIPOLYGON (((-87.592152839 41.816929346, -8... 36
2 MULTIPOLYGON (((-87.628798237 41.801893034, -8... 37
3 MULTIPOLYGON (((-87.606708126 41.816813771, -8... 38
4 MULTIPOLYGON (((-87.592152839 41.816929346, -8... 39

`chicago_names` is DataFrame with community names and area ID
In [12]: chicago_names.head()
Out[12]:
 community NID
0 DOUGLAS 35
1 OAKLAND 36
2 FULLER PARK 37
3 GRAND BOULEVARD 38
4 KENWOOD 39

Merge with `merge` method on shared variable (area ID):
In [13]: chicago_shapes = chicago_shapes.merge(chicago_names, on='NID')

In [14]: chicago_shapes.head()
Out[14]:
 geometry NID community
0 MULTIPOLYGON (((-87.609140876 41.844692503, -8... 35 DOUGLAS
1 MULTIPOLYGON (((-87.592152839 41.816929346, -8... 36 OAKLAND
2 MULTIPOLYGON (((-87.628798237 41.801893034, -8... 37 FULLER PARK
3 MULTIPOLYGON (((-87.606708126 41.816813771, -8... 38 GRAND BOULEVARD
4 MULTIPOLYGON (((-87.592152839 41.816929346, -8... 39 KENWOOD
```

## Spatial joins

In a spatial join, two geometry objects are merged based on their spatial relationship to one another.

```
One GeoDataFrame of communities, one of grocery stores.
```

```
Want to merge to get each grocery's community.
```

```
In [15]: chicago.head()
```

```
Out[15]:
```

		geometry	community
0	MULTIPOLYGON (((1181573.249800048 1886828.0393...,		DOUGLAS
1	MULTIPOLYGON (((1186289.355600054 1876750.7332...,		OAKLAND
2	MULTIPOLYGON (((1176344.998000037 1871187.5456...,		FULLER PARK
3	MULTIPOLYGON (((1182322.042900046 1876674.7304...,	GRAND BOULEVARD	
4	MULTIPOLYGON (((1186289.355600054 1876750.7332...,		KENWOOD

```
In [16]: groceries.head()
```

```
Out[16]:
```

	OBJECTID	...	geometry
0	16	...	MULTIPOINT ((1168268.671671558 1933554.3504257...,
1	18	...	MULTIPOINT ((1162302.617919334 1832900.2240279...,
2	22	...	MULTIPOINT ((1173317.042329894 1895425.4259547...,
3	23	...	MULTIPOINT ((1168996.475130927 1898801.4056401...,
4	27	...	MULTIPOINT ((1176991.988724414 1847262.4228848...,

[5 rows x 8 columns]

```
Execute spatial join
```

```
In [17]: groceries_with_community = groceries.sjoin(chicago, how="inner", predicate='intersects')
```

```
In [18]: groceries_with_community.head()
```

```
Out[18]:
```

	OBJECTID	Ycoord	...	index_right	community
0	16	41.973266	...	30	UPTOWN
1	18	41.696367	...	73	MORGAN PARK
2	22	41.868634	...	28	NEAR WEST SIDE
3	23	41.877590	...	28	NEAR WEST SIDE
4	27	41.737696	...	39	CHATHAM

[5 rows x 10 columns]

GeoPandas provides two spatial-join functions:

- [GeoDataFrame.sjoin\(\)](#): joins based on binary predicates (intersects, contains, etc.)
- [GeoDataFrame.sjoin\\_nearest\(\)](#): joins based on proximity, with the ability to set a maximum search radius.

### Note

For historical reasons, both methods are also available as top-level functions [sjoin\(\)](#) and [sjoin\\_nearest\(\)](#). It is recommended to use methods as the functions may be deprecated in the future.

## Binary predicate joins

Binary predicate joins are available via [GeoDataFrame.sjoin\(\)](#).

[GeoDataFrame.sjoin\(\)](#) has two core arguments: [how](#) and [predicate](#).

## **predicate**

The `predicate` argument specifies how GeoPandas decides whether or not to join the attributes of one object to another, based on their geometric relationship.

The values for `predicate` correspond to the names of geometric binary predicates and depend on the spatial index implementation.

The default spatial index in GeoPandas currently supports the following values for `predicate` which are defined in the [Shapely documentation](#):

- *intersects*
- *contains*
- *within*
- *touches*
- *crosses*
- *overlaps*

## **how**

The `how` argument specifies the type of join that will occur and which geometry is retained in the resultant [`GeoDataFrame`](#). It accepts the following options:

- `left`: use the index from the first (or `left_df`) [`GeoDataFrame`](#) that you provide to [`GeoDataFrame.sjoin\(\)`](#); retain only the `left_df` geometry column
- `right`: use index from second (or `right_df`); retain only the `right_df` geometry column
- `inner`: use intersection of index values from both [`GeoDataFrame`](#); retain only the `left_df` geometry column

Note more complicated spatial relationships can be studied by combining geometric operations with spatial join. To find all polygons within a given distance of a point, for example, one can first use the [`buffer\(\)`](#) method to expand each point into a circle of appropriate radius, then intersect those buffered circles with the polygons in question.

## Nearest joins

Proximity-based joins can be done via [`GeoDataFrame.sjoin\_nearest\(\)`](#).

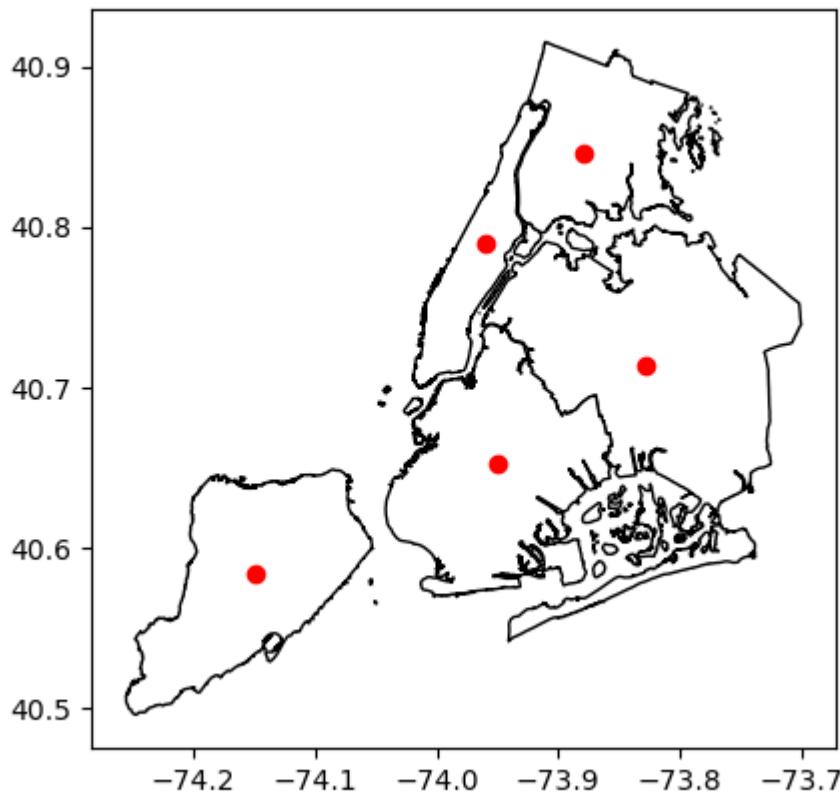
[`GeoDataFrame.sjoin\_nearest\(\)`](#) shares the `how` argument with [`GeoDataFrame.sjoin\(\)`](#), and includes two additional arguments: `max_distance` and `distance_col`.



# Geocoding

GeoPandas supports geocoding (i.e., converting place names to location on Earth) through [geopy](#), an optional dependency of GeoPandas. The following example shows how to get the locations of boroughs in New York City, and plots those locations along with the detailed borough boundary file included within GeoPandas.

```
>>>
In [1]: import geodatasets
In [2]: boros = geopandas.read_file(geodatasets.get_path("nybb"))
In [3]: boros.BoroName
Out[3]:
0 Staten Island
1 Queens
2 Brooklyn
3 Manhattan
4 Bronx
Name: BoroName, dtype: object
In [4]: boro_locations = geopandas.tools.geocode(boros.BoroName)
In [5]: boro_locations
Out[5]:
 geometry address
0 POINT (-74.1496048 40.5834557) Staten Island, New York, New York, United States
1 POINT (-73.8283132 40.7135078) Queens, New York, New York, United States
2 POINT (-73.9497211 40.6526006) Brooklyn, New York, New York, United States
3 POINT (-73.9598939 40.7896239) Manhattan, New York, New York, United States
4 POINT (-73.8785937 40.8466508) The Bronx, New York, New York, United States
In [6]: import matplotlib.pyplot as plt
In [7]: fig, ax = plt.subplots()
In [8]: boros.to_crs("EPSG:4326").plot(ax=ax, color="white", edgecolor="black");
In [9]: boro_locations.plot(ax=ax, color="red");
```



By default, the `geocode()` function uses the [Photon geocoding API](#). But a different geocoding service can be specified with the `provider` keyword.

The argument to `provider` can either be a string referencing geocoding services, such as `'google'`, `'bing'`, `'yahoo'`, and `'openmapquest'`, or an instance of a `Geocoder` from [geopy](#). See [geonav\\_geocoders.SERVICE\\_TO\\_GEOCODER](#) for the full list. For many providers, parameters such as API

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

## Note

This page was generated from [docs/user\\_guide/sampling.ipynb](#).

Interactive online version: [!\[\]\(76d38023a5fd60324c5f081f54334901\_img.jpg\) launch binder](#)

# Sampling Points

Learn how to sample random points using GeoPandas.

The example below shows you how to sample random locations from shapes in GeoPandas GeoDataFrames.

## Import Packages

To begin with, we need to import packages we'll use:

```
[1]: import geopandas
 import geodatasets
```

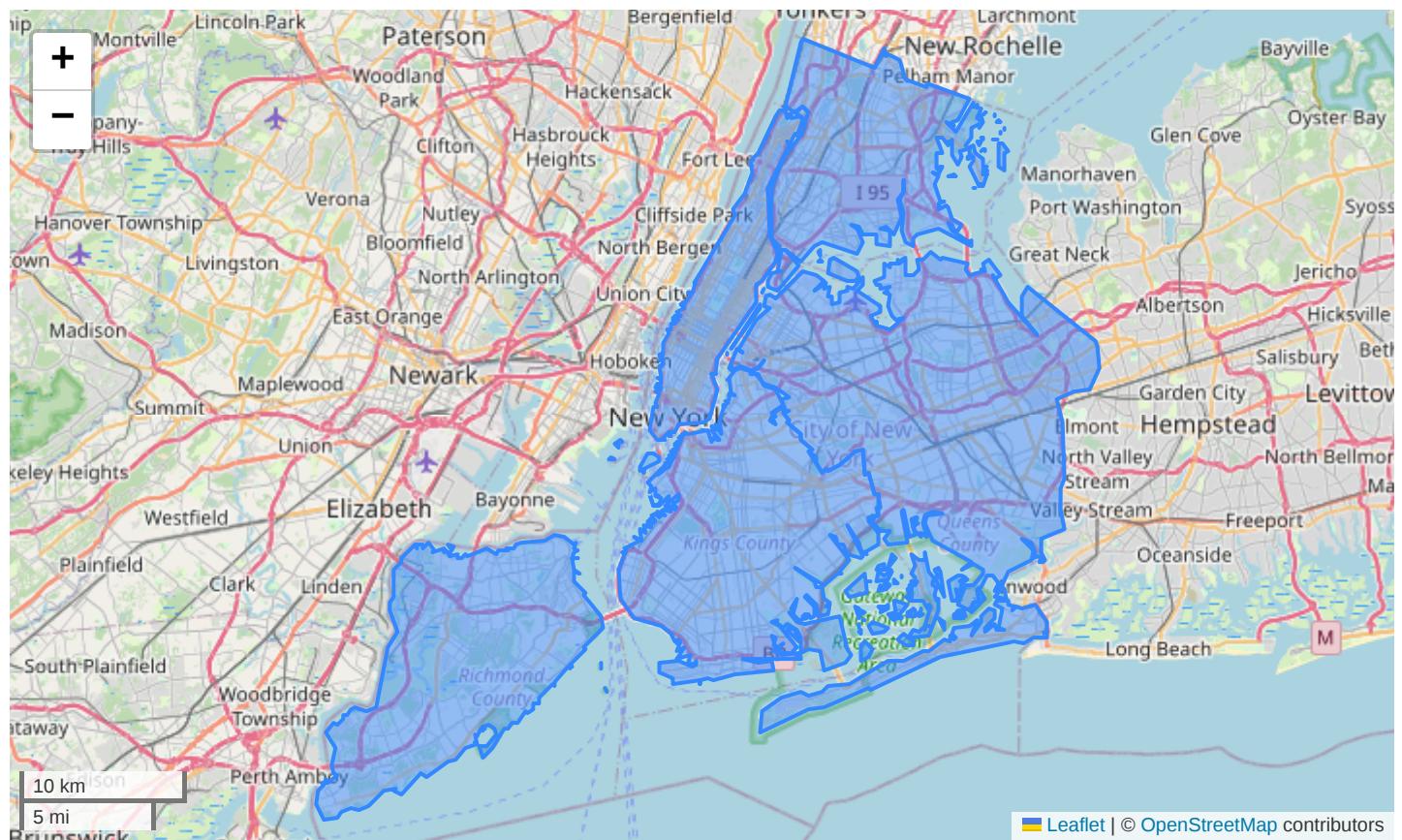
For this example, we will use the New York Borough example data (`nybb`) provided by geodatasets.

```
[2]: nybb = geopandas.read_file(geodatasets.get_path("nybb"))
 # simplify geometry to save space when rendering many interactive maps
 nybb.geometry = nybb.simplify(200)
```

To see what this looks like, visualize the data:

```
[3]: nybb.explore()
```

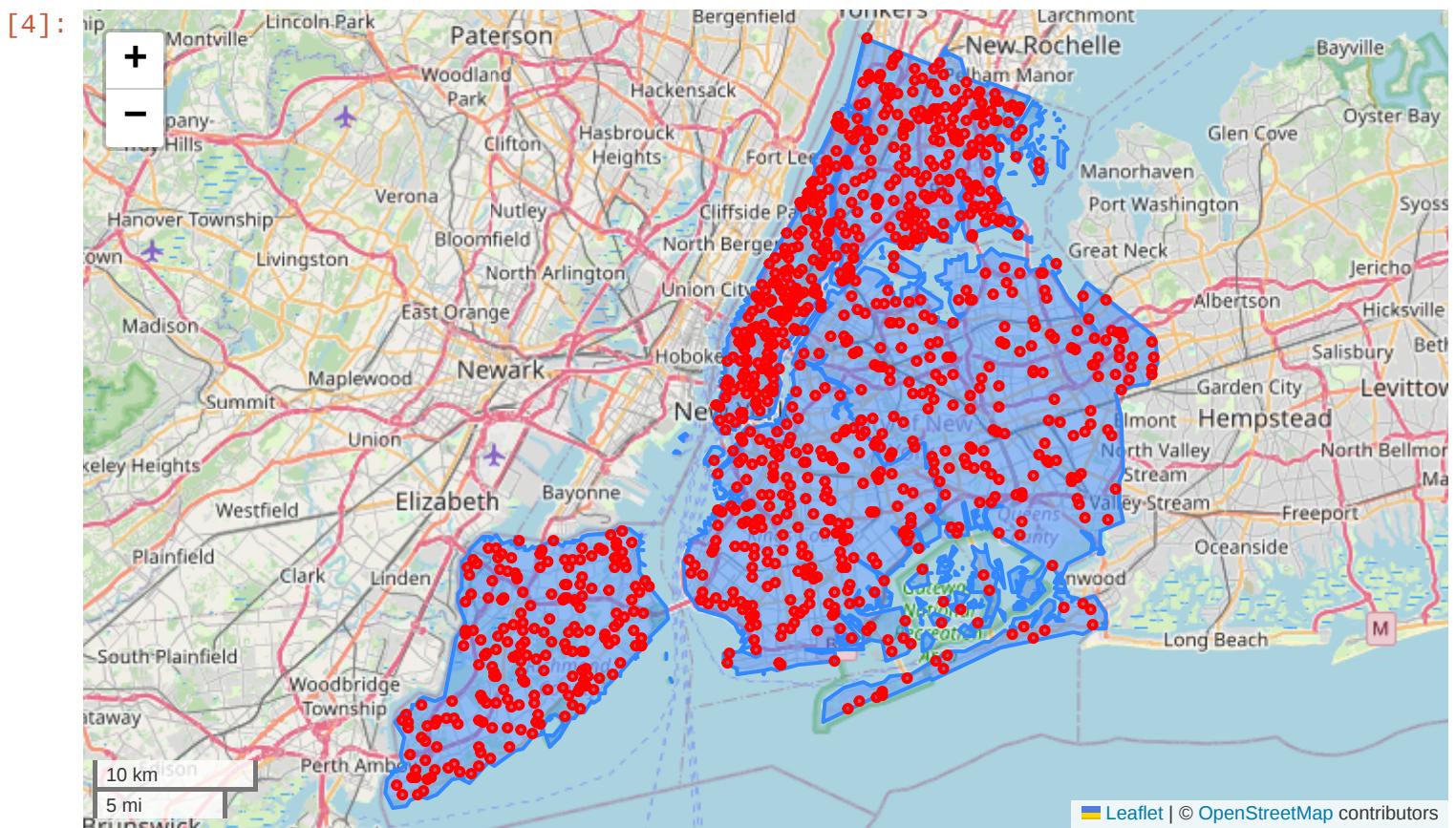
[3]:



## Sampling random points

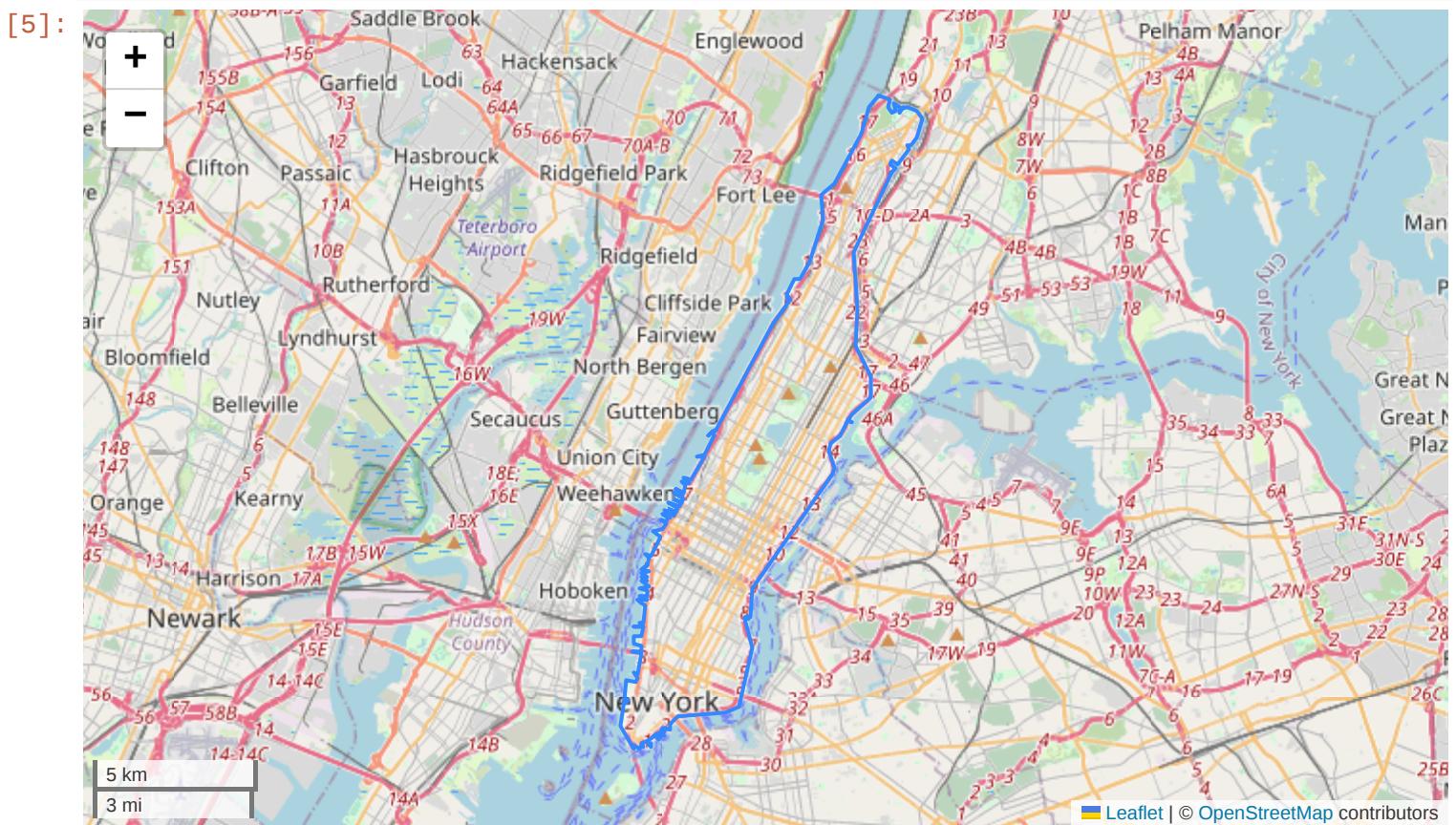
To sample points from within a GeoDataFrame, use the `sample_points()` method. To specify the sample sizes, provide an explicit number of points to sample. For example, we can sample 200 points randomly from each feature:

```
[4]: n200_sampled_points = nybb.sample_points(200)
m = nybb.explore()
n200_sampled_points.explore(m=m, color='red')
```



This functionality also works for line geometries. For example, let's look only at the boundary of Manhattan Island:

```
[5]: manhattan_parts = nybb.iloc[[3]].explode(ignore_index=True)
manhattan_island = manhattan_parts.iloc[[30]]
manhattan_island.boundary.explore()
```



Sampling randomly from along this boundary can use the same `sample_points()` method:

```
[6]: manhattan_border_points = manhattan_island.boundary.sample_points(200)
m = manhattan_island.explore()
manhattan_border_points.explore(m=m, color='red')
```



Keep in mind that sampled points are returned as a single multi-part geometry, and that the distances over the line segments are calculated *along* the line.

```
[7]: manhattan_border_points
```

```
[7]: 30 MULTIPOLY ((978983.931 196716.61), (978989.23...
Name: sampled_points, dtype: geometry
```

If you want to separate out the individual sampled points, use the `.explode()` method on the dataframe:

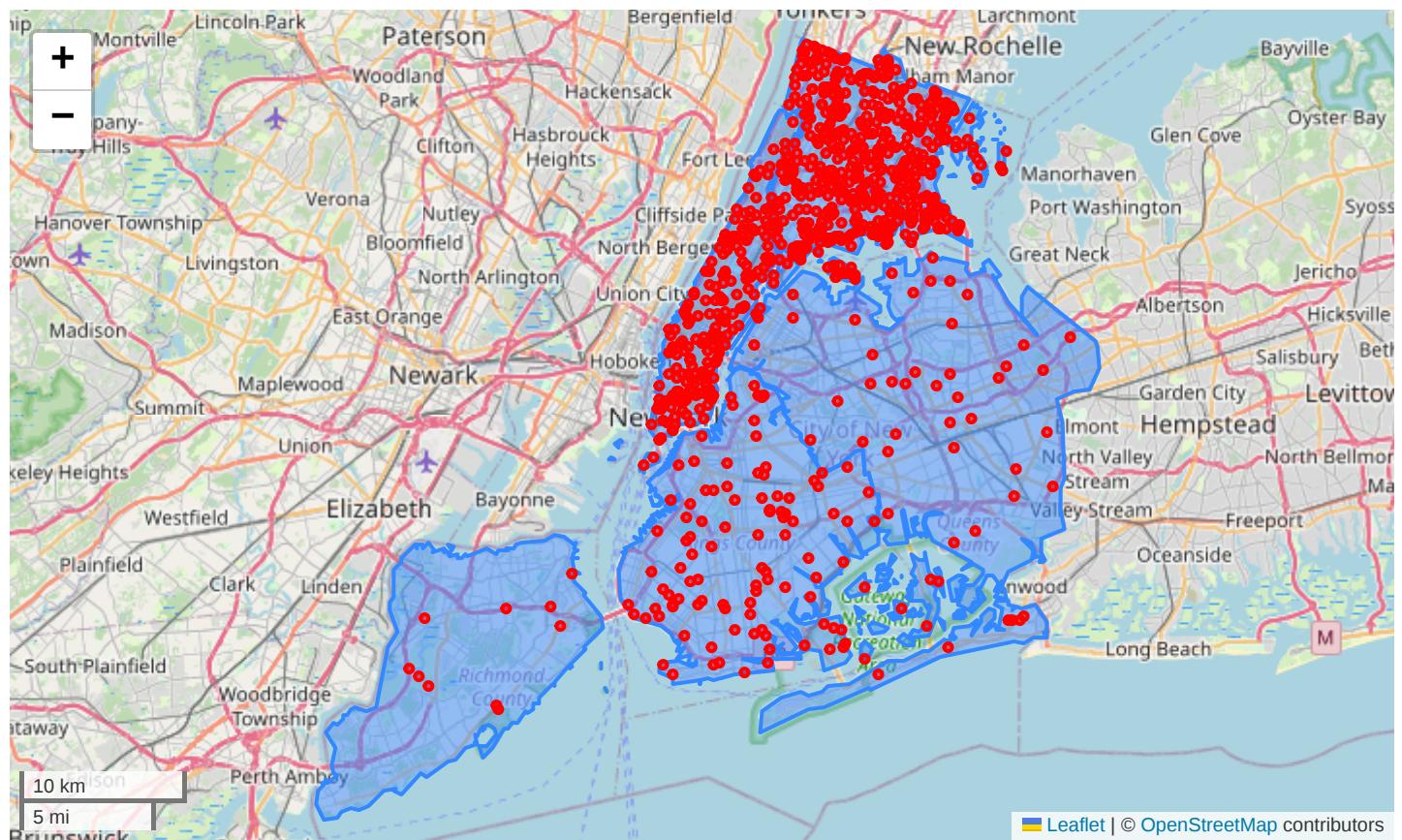
```
[8]: manhattan_border_points.explode(ignore_index=True).head()
```

```
[8]: 0 POINT (978983.931 196716.61)
1 POINT (978989.238 196344.744)
2 POINT (979099.118 196149.568)
3 POINT (979355.019 199203.072)
4 POINT (979591.819 199148.425)
Name: sampled_points, dtype: geometry
```

## Variable number of points

You can also sample different number of points from different geometries if you pass an array specifying the size of the sample per geometry.

```
[9]: variable_size = nybb.sample_points([10, 50, 100, 200, 500])
m = nybb.explore()
variable_size.explore(m=m, color='red')
```



© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# How to...

## Drop duplicate geometry in all situations

Using the standard Pandas [drop\\_duplicates\(\)](#) function on a geometry column can lead to some duplicate geometries not being dropped, in certain circumstances. When used on a geometry column, the Pandas function compares the WKB of each geometry object. This is sensitive to the orders of various components of the geometry - for example, a line with co-ordinates in the order left-to-right should be equal to a line with the same co-ordinates in the order right-to-left, but the WKB representations will be different. The same applies for the order of rings of polygons and parts in multipart geometries.

To deal with this problem, use the [normalize\(\)](#) method first to order the co-ordinates in a canonical form, and then use the standard [drop\\_duplicates\(\)](#) method:

```
gdf["geometry"] = gdf.normalize()
gdf.drop_duplicates()
```

The effect of the [normalize\(\)](#) method can be seen in the following example:

```
>>> geopandas.GeoSeries([
... shapely.LineString([(0, 0), (1, 0), (2, 0)]),
... shapely.LineString([(2, 0), (1, 0), (0, 0)]),
...]).normalize().to_wkt()
0 LINESTRING (0 0, 1 0, 2 0)
1 LINESTRING (0 0, 1 0, 2 0)
dtype: object
```

# Advanced guide

The advanced guide covers advanced usage of GeoPandas. Each page focuses on a single topic and outlines how it is implemented in GeoPandas, with reproducible examples.

If you don't know anything about GeoPandas, start with the [Introduction to GeoPandas](#).

Basic topics can be found in the [User guide](#) and further specification in the [API reference](#).

## Note

This section is currently work in progress. See the available pages below.

## [Missing and empty geometries](#)

[Changes since GeoPandas v0.6.0](#)

## [Re-projecting using GDAL with Rasterio and Fiona](#)

[Fiona example](#)

[Rasterio example](#)

## [Migration from PyGEOS geometry backend to Shapely 2.0](#)

[Migration period](#)

[How to prepare your code for transition](#)

## [Migration from the Fiona to the Pyogrio read/write engine](#)

[Write an attribute table to a file](#)

[No support for `schema` parameter to write files](#)

[Writing EMPTY geometries](#)

# Missing and empty geometries

GeoPandas supports, just like in pandas, the concept of missing values (NA or null values). But for geometry values, there is an additional concept of empty geometries:

- **Empty geometries** are actual geometry objects but that have no coordinates (and thus also no area, for example). They can for example originate from taking the intersection of two polygons that have no overlap. The scalar object (when accessing a single element of a GeoSeries) is still a Shapely geometry object.
- **Missing geometries** are unknown values in a GeoSeries. They will typically be propagated in operations (for example in calculations of the area or of the intersection), or ignored in reductions such as `union_all()`. The scalar object (when accessing a single element of a GeoSeries) is the Python `None` object.

## ⚠ Warning

Starting from GeoPandas v0.6.0, those two concepts are more consistently separated. See [below](#) for more details on what changed compared to earlier versions.

Consider the following example GeoSeries with one polygon, one missing value and one empty polygon:

```
In [1]: from shapely.geometry import Polygon >>>
In [2]: s = geopandas.GeoSeries([Polygon([(0, 0), (1, 1), (0, 1), (0, 0)]), None, Polygon([])])
In [3]: s
Out[3]:
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 None
2 POLYGON EMPTY
dtype: geometry
```

In spatial operations, missing geometries will typically propagate (be missing in the result as well), while empty geometries are treated as a geometry and the result will depend on the operation:

&gt;&gt;&gt;

```
In [4]: s.area
Out[4]:
0 0.5
1 NaN
2 0.0
dtype: float64

In [5]: s.union(Polygon([(0, 0), (0, 1), (1, 1), (1, 0)]))
Out[5]:
0 POLYGON ((1 1, 1 0, 0 0, 0 1, 1 1))
1 None
2 MULTIPOLYGON (EMPTY, ((0 0, 0 1, 1 1, 1 0, 0 0)))
dtype: geometry

In [6]: s.intersection(Polygon([(0, 0), (0, 1), (1, 1), (1, 0)]))
Out[6]:
0 POLYGON ((0 0, 0 1, 1 1, 0 0))
1 None
2 POLYGON EMPTY
dtype: geometry
```

The [`GeoSeries.isna\(\)`](#) method will only check for missing values and not for empty geometries:

```
In [7]: s.isna()
Out[7]:
0 False
1 True
2 False
dtype: bool
```

&gt;&gt;&gt;

On the other hand, if you want to know which values are empty geometries, you can use the [`GeoSeries.is\_empty`](#) attribute:

```
In [8]: s.is_empty
Out[8]:
0 False
1 False
2 True
dtype: bool
```

&gt;&gt;&gt;

To get only the actual geometry objects that are neither missing nor empty, you can use a combination of both:

```
In [9]: s.is_empty | s.isna()
Out[9]:
0 False
1 True
2 True
dtype: bool

In [10]: s[~(s.is_empty | s.isna())]
Out[10]:
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
dtype: geometry
```

&gt;&gt;&gt;

# Changes since GeoPandas v0.6.0

In GeoPandas v0.6.0, the missing data handling was refactored and made more consistent across the library.

Historically, missing (“NA”) values in a GeoSeries could be represented by empty geometric objects, in addition to standard representations such as `None` and `np.nan`. At least, this was the case in `GeoSeries.isna()` or when a GeoSeries got aligned in geospatial operations. But, other methods like `dropna()` and `fillna()` did not follow this approach and did not consider empty geometries as missing.

In GeoPandas v0.6.0, the most important change is `GeoSeries.isna()` no longer treating empty as missing:

- Using the small example from above, the old behaviour treated both the empty as missing geometry as “missing”:

```
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 None
2 GEOMETRYCOLLECTION EMPTY
dtype: object

>>> s.isna()
0 False
1 True
2 True
dtype: bool
```

- Starting from GeoPandas v0.6.0, it will now only see actual missing values as missing:

```
In [11]: s.isna()
Out[11]:
0 False
1 True
2 False
dtype: bool
```

For now, when `isna()` is called on a GeoSeries with empty geometries, a warning is raised to alert the user of the changed behaviour with an indication how to solve this.

Additionally, the behaviour of `GeoSeries.align()` changed to use missing values instead of empty geometries to fill non-matching indexes. Consider the following small toy example:

&gt;&gt;&gt;

```
In [12]: from shapely.geometry import Point
```

```
In [13]: s1 = geopandas.GeoSeries([Point(0, 0), Point(1, 1)], index=[0, 1])
```

```
In [14]: s2 = geopandas.GeoSeries([Point(1, 1), Point(2, 2)], index=[1, 2])
```

```
In [15]: s1
```

```
Out[15]:
```

```
0 POINT (0 0)
1 POINT (1 1)
dtype: geometry
```

```
In [16]: s2
```

```
Out[16]:
```

```
1 POINT (1 1)
2 POINT (2 2)
dtype: geometry
```

- Previously, the `align` method would use empty geometries to fill values:

```
>>> s1_aligned, s2_aligned = s1.align(s2)
```

&gt;&gt;&gt;

```
>>> s1_aligned
0 POINT (0 0)
1 POINT (1 1)
2 GEOMETRYCOLLECTION EMPTY
dtype: object
```

```
>>> s2_aligned
0 GEOMETRYCOLLECTION EMPTY
1 POINT (1 1)
2 POINT (2 2)
dtype: object
```

This method is used under the hood when performing spatial operations on mis-aligned GeoSeries objects:

```
>>> s1.intersection(s2)
0 GEOMETRYCOLLECTION EMPTY
1 POINT (1 1)
2 GEOMETRYCOLLECTION EMPTY
dtype: object
```

&gt;&gt;&gt;

- Starting from GeoPandas v0.6.0, `GeoSeries.align()` will use missing values to fill in the non-aligned indices, to be consistent with the behaviour in pandas:

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# Re-projecting using GDAL with Rasterio and Fiona

The simplest method of re-projecting is `GeoDataFrame.to_crs()`. It uses pyproj as the engine and transforms the points within the geometries.

These examples demonstrate how to use Fiona or rasterio as the engine to re-project your data. Fiona and rasterio are powered by GDAL and with algorithms that consider the geometry instead of just the points the geometry contains. This is particularly useful for antimeridian cutting. However, this also means the transformation is not as fast.

## Fiona example

```
from functools import partial

import fiona
import geopandas
from fiona.transform import transform_geom
from packaging import version
from pyproj import CRS
from pyproj.enums import WktVersion
from shapely.geometry import mapping, shape

set up Fiona transformer
def crs_to_fiona(proj_crs):
 proj_crs = CRS.from_user_input(proj_crs)
 if version.parse(fiona.__gdal_version__) < version.parse("3.0.0"):
 fio_crs = proj_crs.to_wkt(WktVersion.WKT1_GDAL)
 else:
 # GDAL 3+ can use WKT2
 fio_crs = proj_crs.to_wkt()
 return fio_crs

def base_transformer(geom, src_crs, dst_crs):
 return shape(
 transform_geom(
 src_crs=crs_to_fiona(src_crs),
 dst_crs=crs_to_fiona(dst_crs),
 geom=mapping(geom),
 antimeridian_cutting=True,
)
)
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# Migration from PyGEOS geometry backend to Shapely 2.0

Since the 0.8 version, GeoPandas includes an experimental support of PyGEOS as an alternative geometry backend to Shapely. Recently, PyGEOS codebase was merged into the Shapely project and released as part of Shapely 2.0. GeoPandas will therefore deprecate support of the PyGEOS backend and will go forward with Shapely 2.0 as the only geometry engine exposing GEOS functionality.

Given that historically the PyGEOS engine was automatically used if the package is installed (this behaviour will change in GeoPandas 0.14 where Shapely 2.0 is used by default if installed), some downstream code may depend on PyGEOS geometries being available as underlying data of a `GeometryArray`.

This guide outlines the migration from the PyGEOS-based code to the Shapely-based code.

## Migration period

The migration is planned for three releases spanning approximately one year, starting with 0.13 released in the second quarter of 2023.

### GeoPandas 0.13

- PyGEOS is still used as a default backend over Shapely (1.8 or 2.0) if installed, with a `FutureWarning` warning about upcoming changes.

### GeoPandas 0.14

- The default backend is Shapely 2.0 and the PyGEOS is used only if Shapely 1.8 is installed instead of 2.0 or newer. The PyGEOS backend is still supported, but a user needs to opt in using the environment variable `USE_PYGEOS` as explained in the [installation instructions](#).

### GeoPandas 1.0

- GeoPandas will remove support of both PyGEOS and Shapely<2.

## How to prepare your code for transition

If you don't use PyGEOS explicitly, there nothing to be done as GeoPandas internals will take care of the transition. If you use PyGEOS directly and access an array of PyGEOS geometries using `GeoSeries.values.data`, you will need to make some changes to avoid code breakage.

The recommended way is using Shapely vectorized operations on the `GeometryArray` instead of accessing the NumPy array of geometries and using PyGEOS/Shapely operations on the array.

This is a common pattern used with GeoPandas 0.12 (or earlier), that should now be avoided in new code:

```
>>> import pygeos
>>> geometries = gdf.geometry.values.data
>>> mrr = pygeos.minimum_rotated_rectangle(geometries)
```

The recommended way of refactoring this code would look like this (with Geopandas 0.12 or later):

```
>>> import shapely # shapely 2.0
>>> mrr = shapely.minimum_rotated_rectangle(gdf.geometry.array)
```

This code will work no matter which geometry backend GeoPandas actually uses, because on the `GeometryArray` level, it always returns Shapely geometry. Although keep in mind, that it may involve additional overhead cost of converting PyGEOS geometry to Shapely geometry.

Note that while in most cases, a simple replacement of `pygeos` with `shapely` together with a change of `gdf.geometry.values.data` to `gdf.geometry.values` or analogous `gdf.geometry.array` should work, there are some differences between the API of PyGEOS and that of Shapely. Please consult the [Migrating from PyGEOS](#) document for details.

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx 7.3.7](#).

# Migration from the Fiona to the Pyogrio read/write engine

Since version 0.11, GeoPandas started supporting two engines to read and write files: [Fiona](#) and [Pyogrio](#).

It became possible to choose the engine using the `engine=` parameter in [`geopandas.read\_file\(\)`](#) and [`geopandas.GeoDataFrame.to\_file\(\)`](#). It became also possible to change the default engine globally with:

```
geopandas.options.io_engine = "pyogrio"
```

For Geopandas versions <1.0, GeoPandas defaulted to use Fiona. Starting from GeoPandas version 1.0, the global default has changed from Fiona to Pyogrio.

The main reason for this change is performance. Pyogrio is optimized for the use case relevant for GeoPandas: reading and writing in bulk. Because of this, in many cases speedups >5-20x can be observed.

This guide outlines the (known) functional differences between both, so you can account for them when switching to Pyogrio.

## Write an attribute table to a file

Using the Fiona engine, it was possible to write an attribute table (a table without geometry column) to a file using the `schema` parameter to specify that the “geometry” column of a GeoDataFrame should be ignored.

With Pyogrio you can write an attribute table by using [`pyogrio.write\_dataframe\(\)`](#) and passing a pandas DataFrame to it:

```
>>> import pyogrio
>>> df = pd.DataFrame({"data_column": [1, 2, 3]})
>>> pyogrio.write_dataframe(df, "test_attribute_table.gpkg")
```

## No support for `schema` parameter to write files

Pyogrio does not support specifying the `schema` parameter to write files. This means it is not possible to specify the types of attributes being written explicitly.

## Writing EMPTY geometries

```
>>>
In [1]: import shapely
In [2]: gdf = geopandas.GeoDataFrame(geometry=[shapely.Polygon(), None], crs=31370)
In [3]: gdf.to_file("test_fiona.gpkg", engine="fiona")
In [4]: gdf.to_file("test_pyogrio.gpkg", engine="pyogrio")
In [5]: geopandas.read_file("test_fiona.gpkg").head()
Out[5]:
 geometry
0 None
1 None

In [6]: geopandas.read_file("test_pyogrio.gpkg").head()
Out[6]:
 geometry
0 POLYGON EMPTY
1 None
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# GeoSeries

## Constructor

`GeoSeries([data, index, crs])`

A Series object designed to store shapely geometry objects.

## General methods and attributes

<code>GeoSeries.area</code>	Returns a <code>Series</code> containing the area of each geometry in the <code>GeoSeries</code> expressed in the units of the CRS.
<code>GeoSeries.boundary</code>	Returns a <code>GeoSeries</code> of lower dimensional objects representing each geometry's set-theoretic <i>boundary</i> .
<code>GeoSeries.bounds</code>	Returns a <code>DataFrame</code> with columns <code>minx</code> , <code>miny</code> , <code>maxx</code> , <code>maxy</code> values containing the bounds for each geometry.
<code>GeoSeries.total_bounds</code>	Returns a tuple containing <code>minx</code> , <code>miny</code> , <code>maxx</code> , <code>maxy</code> values for the bounds of the series as a whole.
<code>GeoSeries.length</code>	Returns a <code>Series</code> containing the length of each geometry expressed in the units of the CRS.
<code>GeoSeries.geom_type</code>	Returns a <code>Series</code> of strings specifying the <i>Geometry Type</i> of each object.
<code>GeoSeries.offset_curve</code> (distance[, ...])	Returns a <code>LineString</code> or <code>MultiLineString</code> geometry at a distance from the object on its right or its left side.
<code>GeoSeries.distance</code> (other[, align])	Returns a <code>Series</code> containing the distance to aligned <i>other</i> .
<code>GeoSeries.hausdorff_distance</code> (other[, align, ...])	Returns a <code>Series</code> containing the Hausdorff distance to aligned <i>other</i> .

<a href="#"><code>GeoSeries.frechet_distance</code></a> (other[, align, ...])	Returns a <a href="#">Series</a> containing the Frechet distance to aligned <i>other</i> .
<a href="#"><code>GeoSeries.representative_point</code></a> ()	Returns a <a href="#">GeoSeries</a> of (cheaply computed) points that are guaranteed to be within each geometry.
<a href="#"><code>GeoSeries.exterior</code></a>	Returns a <a href="#">GeoSeries</a> of LinearRings representing the outer boundary of each polygon in the GeoSeries.
<a href="#"><code>GeoSeries.interiors</code></a>	Returns a <a href="#">Series</a> of List representing the inner rings of each polygon in the GeoSeries.
<a href="#"><code>GeoSeries.minimum_bounding_radius</code></a> ()	Returns a Series of the radii of the minimum bounding circles that enclose each geometry.
<a href="#"><code>GeoSeries.x</code></a>	Return the x location of point geometries in a GeoSeries
<a href="#"><code>GeoSeries.y</code></a>	Return the y location of point geometries in a GeoSeries
<a href="#"><code>GeoSeries.z</code></a>	Return the z location of point geometries in a GeoSeries
<a href="#"><code>GeoSeries.get_coordinates</code></a> ([include_z, ...])	Gets coordinates from a <a href="#">GeoSeries</a> as a <a href="#">DataFrame</a> of floats.
<a href="#"><code>GeoSeries.count_coordinates</code></a> ()	Returns a <a href="#">Series</a> containing the count of the number of coordinate pairs in each geometry.
<a href="#"><code>GeoSeries.count_geometries</code></a> ()	Returns a <a href="#">Series</a> containing the count of geometries in each multi-part geometry.
<a href="#"><code>GeoSeries.count_interior_rings</code></a> ()	Returns a <a href="#">Series</a> containing the count of the number of interior rings in a polygonal geometry.
<a href="#"><code>GeoSeries.set_precision</code></a> (grid_size[, mode])	Returns a <a href="#">GeoSeries</a> with the precision set to a precision grid size.
<a href="#"><code>GeoSeries.get_precision</code></a> ()	Returns a <a href="#">Series</a> of the precision of each geometry.
<a href="#"><code>GeoSeries.get_geometry</code></a> (index)	Returns the n-th geometry from a collection of geometries.

# Unary predicates

<code>GeoSeries.is_closed</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> if a LineString's or LinearRing's first and last points are equal.
<code>GeoSeries.is_empty</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for empty geometries.
<code>GeoSeries.is_ring</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for features that are closed.
<code>GeoSeries.is_simple</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for geometries that do not cross themselves.
<code>GeoSeries.is_valid</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for geometries that are valid.
<code>GeoSeries.is_valid_reason()</code>	Returns a <code>Series</code> of strings with the reason for invalidity of each geometry.
<code>GeoSeries.has_z</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for features that have a z-component.
<code>GeoSeries.is_ccw</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> if a LineString or LinearRing is counterclockwise.

# Binary predicates

<code>GeoSeries.contains</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that contains <code>other</code> .
<code>GeoSeries.contains_properly</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is completely inside <code>other</code> , with no common boundary points.
<code>GeoSeries.crosses</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that cross <code>other</code> .
<code>GeoSeries.disjoint</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value

`True` for each aligned geometry disjoint to *other*.

`GeoSeries.dwithin`(*other*, *distance*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is within a set distance from *other*.

`GeoSeries.geom_equals`(*other*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry equal to *other*.

`GeoSeries.geom_almost_equals`(*other*[, ...])

Returns a `Series` of `dtype('bool')` with value `True` if each aligned geometry is approximately equal to *other*.

`GeoSeries.geom_equals_exact`(*other*, *tolerance*)

Return True for all geometries that equal aligned *other* to a given tolerance, else False.

`GeoSeries.intersects`(*other*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that intersects *other*.

`GeoSeries.overlaps`(*other*[, *align*])

Returns True for all aligned geometries that overlap *other*, else False.

`GeoSeries.touches`(*other*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that touches *other*.

`GeoSeries.within`(*other*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is within *other*.

`GeoSeries.covers`(*other*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is entirely covering *other*.

`GeoSeries.covered_by`(*other*[, *align*])

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is entirely covered by *other*.

`GeoSeries.relate`(*other*[, *align*])

Returns the DE-9IM intersection matrices for the geometries

`GeoSeries.relate_pattern`(*other*, *pattern*[, *align*])

Returns True if the DE-9IM string code for the relationship between the geometries satisfies the pattern, else False.

# Set-theoretic methods

<code>GeoSeries.clip_by_rect</code> (xmin, ymin, xmax, ymax)	Returns a <code>GeoSeries</code> of the portions of geometry within the given rectangle.
<code>GeoSeries.difference</code> (other[, align])	Returns a <code>GeoSeries</code> of the points in each aligned geometry that are not in <i>other</i> .
<code>GeoSeries.intersection</code> (other[, align])	Returns a <code>GeoSeries</code> of the intersection of points in each aligned geometry with <i>other</i> .
<code>GeoSeries.symmetric_difference</code> (other[, align])	Returns a <code>GeoSeries</code> of the symmetric difference of points in each aligned geometry with <i>other</i> .
<code>GeoSeries.union</code> (other[, align])	Returns a <code>GeoSeries</code> of the union of points in each aligned geometry with <i>other</i> .

# Constructive methods and attributes

<code>GeoSeries.boundary</code>	Returns a <code>GeoSeries</code> of lower dimensional objects representing each geometry's set-theoretic <i>boundary</i> .
<code>GeoSeries.buffer</code> (distance[, resolution, ...])	Returns a <code>GeoSeries</code> of geometries representing all points within a given <code>distance</code> of each geometric object.
<code>GeoSeries.centroid</code>	Returns a <code>GeoSeries</code> of points representing the centroid of each geometry.
<code>GeoSeries.concave_hull</code> ([ratio, allow_holes])	Returns a <code>GeoSeries</code> of geometries representing the concave hull of each geometry.
<code>GeoSeries.convex_hull</code>	Returns a <code>GeoSeries</code> of geometries representing the convex hull of each geometry.
<code>GeoSeries.envelope</code>	Returns a <code>GeoSeries</code> of geometries representing the envelope of each geometry.
<code>GeoSeries.extract_unique_points</code> ()	Returns a <code>GeoSeries</code> of MultiPoints representing

	all distinct vertices of an input geometry.
<code>GeoSeries.force_2d()</code>	Forces the dimensionality of a geometry to 2D.
<code>GeoSeries.force_3d([z])</code>	Forces the dimensionality of a geometry to 3D.
<code>GeoSeries.make_valid()</code>	Repairs invalid geometries.
<code>GeoSeries.minimum_bounding_circle()</code>	Returns a <code>GeoSeries</code> of geometries representing the minimum bounding circle that encloses each geometry.
<code>GeoSeries.minimum_clearance()</code>	Returns a <code>Series</code> containing the minimum clearance distance, which is the smallest distance by which a vertex of the geometry could be moved to produce an invalid geometry.
<code>GeoSeries.minimum_rotated_rectangle()</code>	Returns a <code>GeoSeries</code> of the general minimum bounding rectangle that contains the object.
<code>GeoSeries.normalize()</code>	Returns a <code>GeoSeries</code> of normalized geometries to normal form (or canonical form).
<code>GeoSeries.remove_repeated_points([tolerance])</code>	Returns a <code>GeoSeries</code> containing a copy of the input geometry with repeated points removed.
<code>GeoSeries.reverse()</code>	Returns a <code>GeoSeries</code> with the order of coordinates reversed.
<code>GeoSeries.sample_points(size[, method, ...])</code>	Sample points from each geometry.
<code>GeoSeries.segmentize(max_segment_length)</code>	Returns a <code>GeoSeries</code> with vertices added to line segments based on maximum segment length.
<code>GeoSeries.shortest_line(other[, align])</code>	Returns the shortest two-point line between two geometries.
<code>GeoSeries.simplify(tolerance[, ...])</code>	Returns a <code>GeoSeries</code> containing a simplified representation of each geometry.
<code>GeoSeries.snap(other, tolerance[, align])</code>	Snaps an input geometry to reference geometry's vertices.
<code>GeoSeries.transform(transformation[, include_z])</code>	Returns a <code>GeoSeries</code> with the transformation function applied to the geometry coordinates.

# Affine transformations

<code>GeoSeries.affine_transform</code> (matrix)	Return a <code>GeoSeries</code> with translated geometries.
<code>GeoSeries.rotate</code> (angle[, origin, use_radians])	Returns a <code>GeoSeries</code> with rotated geometries.
<code>GeoSeries.scale</code> ([xfact, yfact, zfact, origin])	Returns a <code>GeoSeries</code> with scaled geometries.
<code>GeoSeries.skew</code> ([xs, ys, origin, use_radians])	Returns a <code>GeoSeries</code> with skewed geometries.
<code>GeoSeries.translate</code> ([xoff, yoff, zoff])	Returns a <code>GeoSeries</code> with translated geometries.

# Linestring operations

<code>GeoSeries.interpolate</code> (distance[, normalized])	Return a point at the specified distance along each geometry
<code>GeoSeries.line_merge</code> ([directed])	Returns (Multi)LineStrings formed by combining the lines in a MultiLineString.
<code>GeoSeries.project</code> (other[, normalized, align])	Return the distance along each geometry nearest to <i>other</i>
<code>GeoSeries.shared_paths</code> (other[, align])	Returns the shared paths between two geometries.

# Aggregating and exploding

<code>GeoSeries.build_area</code> ([node])	Creates an areal geometry formed by the constituent linework.
<code>GeoSeries.delaunay_triangles</code> ([tolerance, ...])	Returns a <code>GeoSeries</code> consisting of objects representing the computed Delaunay triangulation between the vertices of an input geometry.
<code>GeoSeries.explode</code> ([ignore_index, index_parts])	Explode multi-part geometries into multiple single geometries.
<code>GeoSeries.intersection_all</code> ()	Returns a geometry containing the intersection of all geometries in the <code>GeoSeries</code> .

[\*\*GeoSeries.polygonize\*\*](#) ([node, full])

Creates polygons formed from the linework of a GeoSeries.

[\*\*GeoSeries.union\\_all\*\*](#) ([method])

Returns a geometry containing the union of all geometries in the [GeoSeries](#).

[\*\*GeoSeries.voronoi\\_polygons\*\*](#) ([tolerance, ...])

Returns a [GeoSeries](#) consisting of objects representing the computed Voronoi diagram around the vertices of an input geometry.

## Serialization / IO / conversion

[\*\*GeoSeries.from\\_arrow\*\*](#) (arr, \*\*kwargs)

Construct a GeoSeries from a Arrow array object with a GeoArrow extension type.

[\*\*GeoSeries.from\\_file\*\*](#) (filename, \*\*kwargs)

Alternate constructor to create a [GeoSeries](#) from a file.

[\*\*GeoSeries.from\\_wkb\*\*](#) (data[, index, crs, ...])

Alternate constructor to create a [GeoSeries](#) from a list or array of WKB objects

[\*\*GeoSeries.from\\_wkt\*\*](#) (data[, index, crs, ...])

Alternate constructor to create a [GeoSeries](#) from a list or array of WKT objects

[\*\*GeoSeries.from\\_xy\*\*](#) (x, y[, z, index, crs])

Alternate constructor to create a [GeoSeries](#) of Point geometries from lists or arrays of x, y, z) coordinates

[\*\*GeoSeries.to\\_arrow\*\*](#) ([geometry\_encoding, ...])

Encode a GeoSeries to GeoArrow format.

[\*\*GeoSeries.to\\_file\*\*](#) (filename[, driver, index])

Write the [GeoSeries](#) to a file.

[\*\*GeoSeries.to\\_json\*\*](#) ([show\_bbox, drop\_id, to\_wgs84])

Returns a GeoJSON string representation of the GeoSeries.

[\*\*GeoSeries.to\\_wkb\*\*](#) ([hex])

Convert GeoSeries geometries to WKB

[\*\*GeoSeries.to\\_wkt\*\*](#) (\*\*kwargs)

Convert GeoSeries geometries to WKT

## Projection handling

## [GeoSeries.crs](#)

The Coordinate Reference System (CRS) represented as a [pyproj.CRS](#) object.

[GeoSeries.set\\_crs](#)(\*\*kwargs)

[GeoSeries.to\\_crs](#)([crs, epsg])

Returns a [GeoSeries](#) with all geometries transformed to a new coordinate reference system.

[GeoSeries.estimate\\_utm\\_crs](#)([datum\_name])

Returns the estimated UTM CRS based on the bounds of the dataset.

## Missing values

[GeoSeries.fillna](#)([value, inplace, limit])

Fill NA values with geometry (or geometries).

[GeoSeries.isna](#)()

Detect missing values.

[GeoSeries.notna](#)()

Detect non-missing values.

## Overlay operations

[GeoSeries.clip](#)(mask[, keep\_geom\_type, sort])

Clip points, lines, or polygon geometries to the mask extent.

## Plotting

[GeoSeries.plot](#)(\*args, \*\*kwargs)

Plot a GeoSeries.

[GeoSeries.explore](#)(\*args, \*\*kwargs)

Interactive map based on folium/leaflet.js  
Interactive map based on GeoPandas and folium/leaflet.js

## Spatial index

[GeoSeries.sindex](#)

Generate the spatial index

[GeoSeries.has\\_sindex](#)

Check the existence of the spatial index without generating it.

# Indexing

[GeoSeries.cx](#)

Coordinate based indexer to select by intersection with bounding box.

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries

`class geopandas.GeoSeries(data=None, index=None, crs=None, **kwargs)` [\[source\]](#)

A Series object designed to store shapely geometry objects.

## Parameters:

**data : array-like, dict, scalar value**

The geometries to store in the GeoSeries.

**index : array-like or Index**

The index for the GeoSeries.

**crs : value (optional)**

Coordinate Reference System of the geometry objects. Can be anything accepted by

[`pyproj.CRS.from\_user\_input\(\)`](#), such as an authority string (eg “EPSG:4326”) or a WKT string.

**kwargs**

Additional arguments passed to the Series constructor,

e.g. `name`.

## See also

[GeoDataFrame](#)

[pandas.Series](#)

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)])
>>> s
0 POINT (1 1)
1 POINT (2 2)
2 POINT (3 3)
dtype: geometry
```

```
>>> s = geopandas.GeoSeries(
... [Point(1, 1), Point(2, 2), Point(3, 3)], crs="EPSG:3857"
...)
>>> s.crs
<Projected CRS: EPSG:3857>
Name: WGS 84 / Pseudo-Mercator
Axis Info [cartesian]:
- X[east]: Easting (metre)
- Y[north]: Northing (metre)
Area of Use:
- name: World - 85°S to 85°N
- bounds: (-180.0, -85.06, 180.0, 85.06)
Coordinate Operation:
- name: Popular Visualisation Pseudo-Mercator
- method: Popular Visualisation Pseudo Mercator
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
>>> s = geopandas.GeoSeries(
... [Point(1, 1), Point(2, 2), Point(3, 3)], index=["a", "b", "c"], crs=4326
...)
>>> s
a POINT (1 1)
b POINT (2 2)
c POINT (3 3)
dtype: geometry
```

```
>>> s.crs
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

## [\\_\\_init\\_\\_](#)(*data=None*, *index=None*, *crs=None*, *\*\*kwargs*)

[\[source\]](#)

## Methods

[\\_\\_init\\_\\_](#)([*data*, *index*, *crs*])

[abs](#)()

Return a Series/DataFrame with absolute numeric value of each element.

[add](#)(*other*[, *level*, *fill\_value*, *axis*])

Return Addition of series and other, element-wise (binary operator *add*).

<code>add_prefix</code> (prefix[, axis])	Prefix labels with string <i>prefix</i> .
<code>add_suffix</code> (suffix[, axis])	Suffix labels with string <i>suffix</i> .
<code>affine_transform</code> (matrix)	Return a <code>GeoSeries</code> with translated geometries.
<code>agg</code> ([func, axis])	Aggregate using one or more operations over the specified axis.
<code>aggregate</code> ([func, axis])	Aggregate using one or more operations over the specified axis.
<code>align</code> (other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method.
<code>all</code> ([axis, bool_only, skipna])	Return whether all elements are True, potentially over an axis.
<code>any</code> (*[, axis, bool_only, skipna])	Return whether any element is True, potentially over an axis.
<code>append</code> (*args, **kwargs)	
<code>apply</code> (func[, convert_dtype, args])	One-dimensional ndarray with axis labels (including time series).
<code>argmax</code> ([axis, skipna])	Return int position of the largest value in the Series.
<code>argmin</code> ([axis, skipna])	Return int position of the smallest value in the Series.
<code>argsort</code> ([axis, kind, order, stable])	Return the integer indices that would sort the Series values.
<code>asfreq</code> (freq[, method, how, normalize, ...])	Convert time series to specified frequency.
<code>asof</code> (where[, subset])	Return the last row(s) without any NaNs before <i>where</i> .
<code>astype</code> (dtype[, copy, errors])	Cast a pandas object to a specified dtype <code>dtype</code> .
<code>at_time</code> (time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).

<code>autocorr</code> ([lag])	Compute the lag-N autocorrelation.
<code>backfill</code> (*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
<code>between</code> (left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
<code>between_time</code> (start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill</code> (*[, axis, inplace, limit, limit_area, ...])	Fill NA/NaN values by using the next valid observation to fill the gap.
<code>bool</code> ()	Return the bool of a single element Series or DataFrame.
<code>buffer</code> (distance[, resolution, cap_style, ...])	Returns a <code>GeoSeries</code> of geometries representing all points within a given <code>distance</code> of each geometric object.
<code>build_area</code> ([node])	Creates an areal geometry formed by the constituent linework.
<code>case_when</code> (caselist)	Replace values where the conditions are True.
<code>clip</code> (mask[, keep_geom_type, sort])	Clip points, lines, or polygon geometries to the mask extent.
<code>clip_by_rect</code> (xmin, ymin, xmax, ymax)	Returns a <code>GeoSeries</code> of the portions of geometry within the given rectangle.
<code>combine</code> (other, func[, fill_value])	Combine the Series with a Series or scalar according to <code>func</code> .
<code>combine_first</code> (other)	Update null elements with value in the same location in 'other'.
<code>compare</code> (other[, align_axis, keep_shape, ...])	Compare to another Series and show the differences.
<code>concave_hull</code> ([ratio, allow_holes])	Returns a <code>GeoSeries</code> of geometries representing the concave hull of each geometry.

<code><a href="#">contains</a></code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that contains <i>other</i> .
<code><a href="#">contains_properly</a></code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is completely inside <code>other</code> , with no common boundary points.
<code><a href="#">convert_dtypes</a></code> ([infer_objects, ...])	Convert columns to the best possible dtypes using dtypes supporting <code>pd.NA</code> .
<code><a href="#">copy</a></code> ([deep])	Make a copy of this object's indices and data.
<code><a href="#">corr</a></code> (other[, method, min_periods])	Compute correlation with <i>other</i> Series, excluding missing values.
<code><a href="#">count</a></code> ()	Return number of non-NA/null observations in the Series.
<code><a href="#">count_coordinates</a></code> ()	Returns a <code>Series</code> containing the count of the number of coordinate pairs in each geometry.
<code><a href="#">count_geometries</a></code> ()	Returns a <code>Series</code> containing the count of geometries in each multi-part geometry.
<code><a href="#">count_interior_rings</a></code> ()	Returns a <code>Series</code> containing the count of the number of interior rings in a polygonal geometry.
<code><a href="#">cov</a></code> (other[, min_periods, ddof])	Compute covariance with Series, excluding missing values.
<code><a href="#">covered_by</a></code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is entirely covered by <i>other</i> .
<code><a href="#">covers</a></code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is entirely covering <i>other</i> .
<code><a href="#">crosses</a></code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that cross <i>other</i> .

<code>cummax</code> ([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin</code> ([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod</code> ([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
<code>cumsum</code> ([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
<code>delaunay_triangles</code> ([tolerance, only_edges])	Returns a <code>GeoSeries</code> consisting of objects representing the computed Delaunay triangulation between the vertices of an input geometry.
<code>describe</code> ([percentiles, include, exclude])	Generate descriptive statistics.
<code>diff</code> ([periods])	First discrete difference of element.
<code>difference</code> (other[, align])	Returns a <code>GeoSeries</code> of the points in each aligned geometry that are not in <i>other</i> .
<code>disjoint</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry disjoint to <i>other</i> .
<code>distance</code> (other[, align])	Returns a <code>Series</code> containing the distance to aligned <i>other</i> .
<code>div</code> (other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>divide</code> (other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>divmod</code> (other[, level, fill_value, axis])	Return Integer division and modulo of series and other, element-wise (binary operator <i>divmod</i> ).
<code>dot</code> (other)	Compute the dot product between the Series and the columns of other.
<code>drop</code> ([labels, axis, index, columns, level, ...])	Return Series with specified index labels removed.

<code>drop_duplicates</code> (*[, keep, inplace, ignore_index])	Return Series with duplicate values removed.
<code>droplevel</code> (level[, axis])	Return Series/DataFrame with requested index / column level(s) removed.
<code>dropna</code> (*[, axis, inplace, how, ignore_index])	Return a new Series with missing values removed.
<code>duplicated</code> ([keep])	Indicate duplicate Series values.
<code>dwithin</code> (other, distance[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is within a set distance from <code>other</code> .
<code>eq</code> (other[, level, fill_value, axis])	Return Equal to of series and other, element-wise (binary operator <code>eq</code> ).
<code>equals</code> (other)	Test whether two objects contain the same elements.
<code>estimate_utm_crs</code> ([datum_name])	Returns the estimated UTM CRS based on the bounds of the dataset.
<code>ewm</code> ([com, span, halflife, alpha, ...])	Provide exponentially weighted (EW) calculations.
<code>expanding</code> ([min_periods, axis, method])	Provide expanding window calculations.
<code>explode</code> ([ignore_index, index_parts])	Explode multi-part geometries into multiple single geometries.
<code>explore</code> (*args, **kwargs)	Interactive map based on folium/leaflet.js Interactive map based on GeoPandas and folium/leaflet.js
<code>extract_unique_points</code> ()	Returns a <code>GeoSeries</code> of MultiPoints representing all distinct vertices of an input geometry.
<code>factorize</code> ([sort, use_na_sentinel])	Encode the object as an enumerated type or categorical variable.
<code>ffill</code> (*[, axis, inplace, limit, limit_area, ...])	Fill NA/NaN values by propagating the last valid observation to next valid.
<code>fillna</code> ([value, inplace, limit])	Fill NA values with geometry (or geometries).

<code>filter</code> ([items, like, regex, axis])	Subset the dataframe rows or columns according to the specified index labels.
<code>first</code> (offset)	Select initial periods of time series data based on a date offset.
<code>first_valid_index</code> ()	Return index for first non-NA value or None, if no non-NA value is found.
<code>floordiv</code> (other[, level, fill_value, axis])	Return Integer division of series and other, element-wise (binary operator <i>floordiv</i> ).
<code>force_2d</code> ()	Forces the dimensionality of a geometry to 2D.
<code>force_3d</code> ([z])	Forces the dimensionality of a geometry to 3D.
<code>frechet_distance</code> (other[, align, densify])	Returns a <code>Series</code> containing the Frechet distance to aligned <i>other</i> .
<code>from_arrow</code> (arr, **kwargs)	Construct a GeoSeries from a Arrow array object with a GeoArrow extension type.
<code>from_file</code> (filename, **kwargs)	Alternate constructor to create a <code>GeoSeries</code> from a file.
<code>from_wkb</code> (data[, index, crs, on_invalid])	Alternate constructor to create a <code>GeoSeries</code> from a list or array of WKB objects
<code>from_wkt</code> (data[, index, crs, on_invalid])	Alternate constructor to create a <code>GeoSeries</code> from a list or array of WKT objects
<code>from_xy</code> (x, y[, z, index, crs])	Alternate constructor to create a <code>GeoSeries</code> of Point geometries from lists or arrays of x, y(, z) coordinates
<code>ge</code> (other[, level, fill_value, axis])	Return Greater than or equal to of series and other, element-wise (binary operator <i>ge</i> ).
<code>geom_almost_equals</code> (other[, decimal, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> if each aligned geometry is approximately equal to <i>other</i> .
<code>geom_equals</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry equal

	<i>to other.</i>
<a href="#"><b>geom_equals_exact</b></a> (other, tolerance[, align])	Return True for all geometries that equal aligned <i>other</i> to a given tolerance, else False.
<a href="#"><b>get</b></a> (key[, default])	Get item from object for given key (ex: DataFrame column).
<a href="#"><b>get_coordinates</b></a> ([include_z, ignore_index, ...])	Gets coordinates from a <a href="#"><b>GeoSeries</b></a> as a <a href="#"><b>DataFrame</b></a> of floats.
<a href="#"><b>get_geometry</b></a> (index)	Returns the n-th geometry from a collection of geometries.
<a href="#"><b>get_precision</b></a> ()	Returns a <a href="#"><b>Series</b></a> of the precision of each geometry.
<a href="#"><b>groupby</b></a> ([by, axis, level, as_index, sort, ...])	Group Series using a mapper or by a Series of columns.
<a href="#"><b>gt</b></a> (other[, level, fill_value, axis])	Return Greater than of series and other, element-wise (binary operator <i>gt</i> ).
<a href="#"><b>hausdorff_distance</b></a> (other[, align, densify])	Returns a <a href="#"><b>Series</b></a> containing the Hausdorff distance to aligned <i>other</i> .
<a href="#"><b>head</b></a> ([n])	Return the first <i>n</i> rows.
<a href="#"><b>hilbert_distance</b></a> ([total_bounds, level])	Calculate the distance along a Hilbert curve.
<a href="#"><b>hist</b></a> ([by, ax, grid, xlabelsize, xrot, ...])	Draw histogram of the input series using matplotlib.
<a href="#"><b>idxmax</b></a> ([axis, skipna])	Return the row label of the maximum value.
<a href="#"><b>idxmin</b></a> ([axis, skipna])	Return the row label of the minimum value.
<a href="#"><b>infer_objects</b></a> ([copy])	Attempt to infer better dtypes for object columns.
<a href="#"><b>info</b></a> ([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a Series.
<a href="#"><b>interpolate</b></a> (distance[, normalized])	Return a point at the specified distance along each geometry
<a href="#"><b>intersection</b></a> (other[, align])	Returns a <a href="#"><b>GeoSeries</b></a> of the intersection of points in each aligned geometry with <i>other</i> .

<code><u>intersection_all</u>()</code>	Returns a geometry containing the intersection of all geometries in the <code>GeoSeries</code> .
<code><u>intersects</u>(other[, align])</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that intersects <code>other</code> .
<code><u>is_valid_reason</u>()</code>	Returns a <code>Series</code> of strings with the reason for invalidity of each geometry.
<code><u>isin</u>(values)</code>	Whether elements in Series are contained in <code>values</code> .
<code><u>isna</u>()</code>	Detect missing values.
<code><u>isnull</u>()</code>	Alias for <code>isna</code> method.
<code><u>item</u>()</code>	Return the first element of the underlying data as a Python scalar.
<code><u>items</u>()</code>	Lazily iterate over (index, value) tuples.
<code><u>keys</u>()</code>	Return alias for index.
<code><u>kurt</u>([axis, skipna, numeric_only])</code>	Return unbiased kurtosis over requested axis.
<code><u>kurtosis</u>([axis, skipna, numeric_only])</code>	Return unbiased kurtosis over requested axis.
<code><u>last</u>(offset)</code>	Select final periods of time series data based on a date offset.
<code><u>last_valid_index</u>()</code>	Return index for last non-NA value or None, if no non-NA value is found.
<code><u>le</u>(other[, level, fill_value, axis])</code>	Return Less than or equal to of series and other, element-wise (binary operator <code>le</code> ).
<code><u>line_merge</u>([directed])</code>	Returns (Multi)LineStrings formed by combining the lines in a MultiLineString.
<code><u>lt</u>(other[, level, fill_value, axis])</code>	Return Less than of series and other, element-wise (binary operator <code>lt</code> ).
<code><u>make_valid</u>()</code>	Repairs invalid geometries.
<code><u>map</u>(arg[, na_action])</code>	Map values of Series according to an input

	mapping or function.
<code>mask</code> (cond[, other, inplace, axis, level])	Replace values where the condition is True.
<code>max</code> ([axis, skipna, numeric_only])	Return the maximum of the values over the requested axis.
<code>mean</code> ([axis, skipna, numeric_only])	Return the mean of the values over the requested axis.
<code>median</code> ([axis, skipna, numeric_only])	Return the median of the values over the requested axis.
<code>memory_usage</code> ([index, deep])	Return the memory usage of the Series.
<code>min</code> ([axis, skipna, numeric_only])	Return the minimum of the values over the requested axis.
<code>minimum_bounding_circle</code> ()	Returns a <code>GeoSeries</code> of geometries representing the minimum bounding circle that encloses each geometry.
<code>minimum_bounding_radius</code> ()	Returns a <i>Series</i> of the radii of the minimum bounding circles that enclose each geometry.
<code>minimum_clearance</code> ()	Returns a <code>Series</code> containing the minimum clearance distance, which is the smallest distance by which a vertex of the geometry could be moved to produce an invalid geometry.
<code>minimum_rotated_rectangle</code> ()	Returns a <code>GeoSeries</code> of the general minimum bounding rectangle that contains the object.
<code>mod</code> (other[, level, fill_value, axis])	Return Modulo of series and other, element-wise (binary operator <i>mod</i> ).
<code>mode</code> ([dropna])	Return the mode(s) of the Series.
<code>mul</code> (other[, level, fill_value, axis])	Return Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>multiply</code> (other[, level, fill_value, axis])	Return Multiplication of series and other, element-wise (binary operator <i>mul</i> ).

<code>ne</code> (other[, level, fill_value, axis])	Return Not equal to of series and other, element-wise (binary operator <i>ne</i> ).
<code>nlargest</code> ([n, keep])	Return the largest <i>n</i> elements.
<code>normalize</code> ()	Returns a <a href="#">GeoSeries</a> of normalized geometries to normal form (or canonical form).
<code>notna</code> ()	Detect non-missing values.
<code>notnull</code> ()	Alias for <i>notna</i> method.
<code>nsmallest</code> ([n, keep])	Return the smallest <i>n</i> elements.
<code>nunique</code> ([dropna])	Return number of unique elements in the object.
<code>offset_curve</code> (distance[, quad_segs, ...])	Returns a <a href="#">LineString</a> or <a href="#">MultiLineString</a> geometry at a distance from the object on its right or its left side.
<code>overlaps</code> (other[, align])	Returns True for all aligned geometries that overlap <i>other</i> , else False.
<code>pad</code> (*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
<code>pct_change</code> ([periods, fill_method, limit, freq])	Fractional change between the current and a prior element.
<code>pipe</code> (func, *args, **kwargs)	Apply chainable functions that expect Series or DataFrames.
<code>plot</code> (*args, **kwargs)	Plot a GeoSeries.
<code>Polygonize</code> ([node, full])	Creates polygons formed from the linework of a GeoSeries.
<code>pop</code> (item)	Return item and drops from series.
<code>pow</code> (other[, level, fill_value, axis])	Return Exponential power of series and other, element-wise (binary operator <i>pow</i> ).
<code>prod</code> ([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.

<code>product</code> ([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
<code>project</code> (other[, normalized, align])	Return the distance along each geometry nearest to <i>other</i>
<code>quantile</code> ([q, interpolation])	Return value at the given quantile.
<code>radd</code> (other[, level, fill_value, axis])	Return Addition of series and other, element-wise (binary operator <i>radd</i> ).
<code>rank</code> ([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
<code>ravel</code> ([order])	Return the flattened underlying data as an ndarray or ExtensionArray.
<code>rdiv</code> (other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<code>rdivmod</code> (other[, level, fill_value, axis])	Return Integer division and modulo of series and other, element-wise (binary operator <i>rdivmod</i> ).
<code>reindex</code> ([index, axis, method, copy, level, ...])	Conform Series to new index with optional filling logic.
<code>reindex_like</code> (other[, method, copy, limit, ...])	Return an object with matching indices as other object.
<code>relate</code> (other[, align])	Returns the DE-9IM intersection matrices for the geometries
<code>relate_pattern</code> (other, pattern[, align])	Returns True if the DE-9IM string code for the relationship between the geometries satisfies the pattern, else False.
<code>remove_repeated_points</code> ([tolerance])	Returns a <code>GeoSeries</code> containing a copy of the input geometry with repeated points removed.
<code>rename</code> ([index, axis, copy, inplace, level, ...])	Alter Series index labels or name.
<code>rename_axis</code> ([mapper, index, axis, copy, inplace])	Set the name of the axis for the index or columns.
<code>reorder_levels</code> (order)	Rearrange index levels using input order.

<code>repeat</code> (repeats[, axis])	Repeat elements of a Series.
<code>replace</code> ([to_replace, value, inplace, limit, ...])	Replace values given in <code>to_replace</code> with <code>value</code> .
<code>representative_point</code> ()	Returns a <code>GeoSeries</code> of (cheaply computed) points that are guaranteed to be within each geometry.
<code>resample</code> (rule[, axis, closed, label, ...])	Resample time-series data.
<code>reset_index</code> ([level, drop, name, inplace, ...])	Generate a new DataFrame or Series with the index reset.
<code>reverse</code> ()	Returns a <code>GeoSeries</code> with the order of coordinates reversed.
<code>rfloordiv</code> (other[, level, fill_value, axis])	Return Integer division of series and other, element-wise (binary operator <code>rfloordiv</code> ).
<code>rmod</code> (other[, level, fill_value, axis])	Return Modulo of series and other, element-wise (binary operator <code>rmod</code> ).
<code>rmul</code> (other[, level, fill_value, axis])	Return Multiplication of series and other, element-wise (binary operator <code>rmul</code> ).
<code>rolling</code> (window[, min_periods, center, ...])	Provide rolling window calculations.
<code>rotate</code> (angle[, origin, use_radians])	Returns a <code>GeoSeries</code> with rotated geometries.
<code>round</code> ([decimals])	Round each value in a Series to the given number of decimals.
<code>rpow</code> (other[, level, fill_value, axis])	Return Exponential power of series and other, element-wise (binary operator <code>rpow</code> ).
<code>rsub</code> (other[, level, fill_value, axis])	Return Subtraction of series and other, element-wise (binary operator <code>rsub</code> ).
<code>rtruediv</code> (other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator <code>rtruediv</code> ).
<code>sample</code> ([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
<code>sample_points</code> (size[, method, seed, rng])	Sample points from each geometry.

<code>scale</code> ([xfact, yfact, zfact, origin])	Returns a <code>GeoSeries</code> with scaled geometries.
<code>searchsorted</code> (value[, side, sorter])	Find indices where elements should be inserted to maintain order.
<code>segmentize</code> (max_segment_length)	Returns a <code>GeoSeries</code> with vertices added to line segments based on maximum segment length.
<code>select</code> (*args, **kwargs)	One-dimensional ndarray with axis labels (including time series).
<code>sem</code> ([axis, skipna, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
<code>set_axis</code> (labels, *[, axis, copy])	Assign desired index to given axis.
<code>set_crs</code> (**kwargs)	
<code>set_flags</code> (*[, copy, allows_duplicate_labels])	Return a new object with updated flags.
<code>set_precision</code> (grid_size[, mode])	Returns a <code>GeoSeries</code> with the precision set to a precision grid size.
<code>shared_paths</code> (other[, align])	Returns the shared paths between two geometries.
<code>shift</code> ([periods, freq, axis, fill_value, suffix])	Shift index by desired number of periods with an optional time <i>freq</i> .
<code>shortest_line</code> (other[, align])	Returns the shortest two-point line between two geometries.
<code>simplify</code> (tolerance[, preserve_topology])	Returns a <code>GeoSeries</code> containing a simplified representation of each geometry.
<code>skew</code> ([xs, ys, origin, use_radians])	Returns a <code>GeoSeries</code> with skewed geometries.
<code>snap</code> (other, tolerance[, align])	Snaps an input geometry to reference geometry's vertices.
<code>sort_index</code> (*args, **kwargs)	One-dimensional ndarray with axis labels (including time series).
<code>sort_values</code> (*[, axis, ascending, inplace, ...])	Sort by the values.

<code>squeeze</code> ([axis])	Squeeze 1 dimensional axis objects into scalars.
<code>std</code> ([axis, skipna, ddof, numeric_only])	Return sample standard deviation over requested axis.
<code>sub</code> (other[, level, fill_value, axis])	Return Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>subtract</code> (other[, level, fill_value, axis])	Return Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>sum</code> ([axis, skipna, numeric_only, min_count])	Return the sum of the values over the requested axis.
<code>swapaxes</code> (axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
<code>swaplevel</code> ([i, j, copy])	Swap levels i and j in a <code>MultiIndex</code> .
<code>symmetric_difference</code> (other[, align])	Returns a <code>GeoSeries</code> of the symmetric difference of points in each aligned geometry with <i>other</i> .
<code>tail</code> ([n])	Return the last <i>n</i> rows.
<code>take</code> (*args, **kwargs)	One-dimensional ndarray with axis labels (including time series).
<code>to_arrow</code> ([geometry_encoding, interleaved, ...])	Encode a GeoSeries to GeoArrow format.
<code>to_clipboard</code> (*[, excel, sep])	Copy object to the system clipboard.
<code>to_crs</code> ([crs, epsg])	Returns a <code>GeoSeries</code> with all geometries transformed to a new coordinate reference system.
<code>to_csv</code> ([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
<code>to_dict</code> (*[, into])	Convert Series to {label -> value} dict or dict-like object.
<code>to_excel</code> (excel_writer, *[, sheet_name, ...])	Write object to an Excel sheet.
<code>to_file</code> (filename[, driver, index])	Write the <code>GeoSeries</code> to a file.

<code>to_frame</code> ([name])	Convert Series to DataFrame.
<code>to_hdf</code> (path_or_buf, *, key[, mode, ...])	Write the contained data to an HDF5 file using HDFStore.
<code>to_json</code> ([show_bbox, drop_id, to_wgs84])	Returns a GeoJSON string representation of the GeoSeries.
<code>to_latex</code> ([buf, columns, header, index, ...])	Render object to a LaTeX tabular, longtable, or nested table.
<code>to_list</code> ()	Return a list of the values.
<code>to_markdown</code> ([buf, mode, index, storage_options])	Print Series in Markdown-friendly format.
<code>to_numpy</code> ([dtype, copy, na_value])	A NumPy ndarray representing the values in this Series or Index.
<code>to_period</code> ([freq, copy])	Convert Series from DatetimeIndex to PeriodIndex.
<code>to_pickle</code> (path, *[, compression, protocol, ...])	Pickle (serialize) object to file.
<code>to_sql</code> (name, con, *[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
<code>to_string</code> ([buf, na_rep, float_format, ...])	Render a string representation of the Series.
<code>to_timestamp</code> ([freq, how, copy])	Cast to DatetimeIndex of Timestamps, at <i>beginning</i> of period.
<code>to_wkb</code> ([hex])	Convert GeoSeries geometries to WKB
<code>to_wkt</code> (**kwargs)	Convert GeoSeries geometries to WKT
<code>to_xarray</code> ()	Return an xarray object from the pandas object.
<code>tolist</code> ()	Return a list of the values.
<code>touches</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that touches <code>other</code> .
<code>transform</code> (transformation[, include_z])	Returns a <code>GeoSeries</code> with the transformation function applied to the geometry coordinates.

<code><u>translate</u></code> ([xoff, yoff, zoff])	Returns a <code>GeoSeries</code> with translated geometries.
<code>transpose</code> (*args, **kwargs)	Return the transpose, which is by definition self.
<code>truediv</code> (other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>truncate</code> ([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
<code>tz_convert</code> (tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
<code>tz_localize</code> (tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
<code><u>union</u></code> (other[, align])	Returns a <code>GeoSeries</code> of the union of points in each aligned geometry with <i>other</i> .
<code><u>union_all</u></code> ([method])	Returns a geometry containing the union of all geometries in the <code>GeoSeries</code> .
<code>unique</code> ()	Return unique values of Series object.
<code>unstack</code> ([level, fill_value, sort])	Unstack, also known as pivot, Series with MultiIndex to produce DataFrame.
<code>update</code> (other)	Modify Series in place using values from passed Series.
<code>value_counts</code> ([normalize, sort, ascending, ...])	Return a Series containing counts of unique values.
<code>var</code> ([axis, skipna, ddof, numeric_only])	Return unbiased variance over requested axis.
<code>view</code> ([dtype])	Create a new view of the Series.
<code><u>voronoi_polygons</u></code> ([tolerance, extend_to, ...])	Returns a <code>GeoSeries</code> consisting of objects representing the computed Voronoi diagram around the vertices of an input geometry.
<code>where</code> (cond[, other, inplace, axis, level])	Replace values where the condition is False.
<code><u>within</u></code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is

within *other*.

<code>xs</code> (key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.
--------------------------------------------------	-------------------------------------------------

## Attributes

<code>T</code>	Return the transpose, which is by definition self.
<code>area</code>	Returns a <code>Series</code> containing the area of each geometry in the <code>GeoSeries</code> expressed in the units of the CRS.
<code>array</code>	The ExtensionArray of the data backing this Series or Index.
<code>at</code>	Access a single value for a row/column label pair.
<code>attrs</code>	Dictionary of global attributes of this dataset.
<code>axes</code>	Return a list of the row axis labels.
<code>boundary</code>	Returns a <code>GeoSeries</code> of lower dimensional objects representing each geometry's set-theoretic <i>boundary</i> .
<code>bounds</code>	Returns a <code>DataFrame</code> with columns <code>minx</code> , <code>miny</code> , <code>maxx</code> , <code>maxy</code> values containing the bounds for each geometry.
<code>centroid</code>	Returns a <code>GeoSeries</code> of points representing the centroid of each geometry.
<code>convex_hull</code>	Returns a <code>GeoSeries</code> of geometries representing the convex hull of each geometry.
<code>crs</code>	The Coordinate Reference System (CRS) represented as a <code>pyproj.CRS</code> object.
<code>cx</code>	Coordinate based indexer to select by intersection with bounding box.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>dtypes</code>	Return the dtype object of the underlying data.
<code>empty</code>	Indicator whether Series/DataFrame is empty.
<code>envelope</code>	Returns a <code>GeoSeries</code> of geometries representing the envelope of each geometry.

<a href="#"><u>exterior</u></a>	Returns a <a href="#">GeoSeries</a> of LinearRings representing the outer boundary of each polygon in the GeoSeries.
<a href="#"><u>flags</u></a>	Get the properties associated with this pandas object.
<a href="#"><u>geom_type</u></a>	Returns a <a href="#">Series</a> of strings specifying the <i>Geometry Type</i> of each object.
<a href="#"><u>geometry</u></a>	
<a href="#"><u>has_sindex</u></a>	Check the existence of the spatial index without generating it.
<a href="#"><u>has_z</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for features that have a z-component.
<a href="#"><u>hasnans</u></a>	Return True if there are any NaNs.
<a href="#"><u>iat</u></a>	Access a single value for a row/column pair by integer position.
<a href="#"><u>iloc</u></a>	Purely integer-location based indexing for selection by position.
<a href="#"><u>index</u></a>	The index (axis labels) of the Series.
<a href="#"><u>interiors</u></a>	Returns a <a href="#">Series</a> of List representing the inner rings of each polygon in the GeoSeries.
<a href="#"><u>is_ccw</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> if a LineString or LinearRing is counterclockwise.
<a href="#"><u>is_closed</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> if a LineString's or LinearRing's first and last points are equal.
<a href="#"><u>is_empty</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for empty geometries.
<a href="#"><u>is_monotonic_decreasing</u></a>	Return boolean if values in the object are monotonically decreasing.
<a href="#"><u>is_monotonic_increasing</u></a>	Return boolean if values in the object are monotonically increasing.
<a href="#"><u>is_ring</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for features that are closed.
<a href="#"><u>is_simple</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for geometries that do not cross themselves.
<a href="#"><u>is_unique</u></a>	Return boolean if values in the object are unique.
<a href="#"><u>is_valid</u></a>	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for



# geopandas.GeoSeries.area

*property* `GeoSeries.area`

[\[source\]](#)

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

## See also

[`GeoSeries.length`](#)

measure length

## Notes

Area may be invalid for a geographic CRS using degrees as units; use [`GeoSeries.to\_crs\(\)`](#) to project geometries to a planar CRS before using this function.

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... Polygon([(10, 0), (10, 5), (0, 0)]),
... Polygon([(0, 0), (2, 2), (2, 0)]),
... LineString([(0, 0), (1, 1), (0, 1)]),
... Point(0, 1)
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 POLYGON ((10 0, 10 5, 0 0, 10 0))
2 POLYGON ((0 0, 2 2, 2 0, 0 0))
3 LINESTRING (0 0, 1 1, 0 1)
4 POINT (0 1)
dtype: geometry
```

```
>>> s.area
0 0.5
1 25.0
2 2.0
3 0.0
4 0.0
dtype: float64
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.boundary

*property* `GeoSeries.boundary`

[\[source\]](#)

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

## See also

[`GeoSeries.exterior`](#)

outer boundary (without interior rings)

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.boundary
0 LINESTRING (0 0, 1 1, 0 1, 0 0)
1 MULTIPOINT ((0 0), (1 0))
2 GEOMETRYCOLLECTION EMPTY
dtype: geometry
```

# geopandas.GeoSeries.bounds

## property GeoSeries.bounds

[\[source\]](#)

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.

## Examples

```
>>> from shapely.geometry import Point, Polygon, LineString
>>> d = {'geometry': [Point(2, 1), Polygon([(0, 0), (1, 1), (1, 0)]),
... LineString([(0, 1), (1, 2)])]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf.bounds
 minx miny maxx maxy
0 2.0 1.0 2.0 1.0
1 0.0 0.0 1.0 1.0
2 0.0 1.0 1.0 2.0
```

You can assign the bounds to the `GeoDataFrame` as:

```
>>> import pandas as pd
>>> gdf = pd.concat([gdf, gdf.bounds], axis=1)
>>> gdf
 geometry minx miny maxx maxy
0 POINT (2 1) 2.0 1.0 2.0 1.0
1 POLYGON ((0 0, 1 1, 1 0, 0 0)) 0.0 0.0 1.0 1.0
2 LINESTRING (0 1, 1 2) 0.0 1.0 1.0 2.0
```

# geopandas.GeoSeries.total\_bounds

*property* `GeoSeries.total_bounds`

[\[source\]](#)

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.

## Examples

```
>>> from shapely.geometry import Point, Polygon, LineString
>>> d = {'geometry': [Point(3, -1), Polygon([(0, 0), (1, 1), (1, 0)]),
... LineString([(0, 1), (1, 2)])]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf.total_bounds
array([0., -1., 3., 2.])
```

# geopandas.GeoSeries.length

**property** `GeoSeries.length`

[\[source\]](#)

Returns a `Series` containing the length of each geometry expressed in the units of the CRS.

In the case of a (Multi)Polygon it measures the length of its exterior (i.e. perimeter).

## See also

[`GeoSeries.area`](#)

measure area of a polygon

## Notes

Length may be invalid for a geographic CRS using degrees as units; use [`GeoSeries.to\_crs\(\)`](#) to project geometries to a planar CRS before using this function.

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> from shapely.geometry import Polygon, LineString, MultiLineString, Point, Geometr
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (1, 1), (0, 1)]),
... LineString([(10, 0), (10, 5), (0, 0)]),
... MultiLineString([(0, 0), (1, 0), (-1, 0), (1, 0)]),
... Polygon([(0, 0), (1, 1), (0, 1)]),
... Point(0, 1),
... GeometryCollection([Point(1, 0), LineString([(10, 0), (10, 5), (0, 0)])])
...]
...)
>>> s
0 LINESTRING (0 0, 1 1, 0 1)
1 LINESTRING (10 0, 10 5, 0 0)
2 MULTILINESTRING ((0 0, 1 0), (-1 0, 1 0))
3 POLYGON ((0 0, 1 1, 0 1, 0 0))
4 POINT (0 1)
5 GEOMETRYCOLLECTION (POINT (1 0), LINESTRING (1...
dtype: geometry
```

```
>>> s.length
0 2.414214
1 16.180340
2 3.000000
3 3.414214
4 0.000000
5 16.180340
dtype: float64
```

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.geom\_type

*property* `GeoSeries.geom_type`

[\[source\]](#)

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

## Examples

```
>>> from shapely.geometry import Point, Polygon, LineString
>>> d = {'geometry': [Point(2, 1), Polygon([(0, 0), (1, 1), (1, 0)]),
... LineString([(0, 0), (1, 1)])]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf.geom_type
0 Point
1 Polygon
2 LineString
dtype: object
```

# geopandas.GeoSeries.offset\_curve

`GeoSeries.offset_curve(distance, quad_segs=8, join_style='round', mitre_limit=5.0)`

[\[source\]](#)

Returns a `LineString` or `MultiLineString` geometry at a distance from the object on its right or its left side.

## Parameters:

`distance : float | array-like`

Specifies the offset distance from the input geometry. Negative for right side offset, positive for left side offset.

`quad_segs : int (optional, default 8)`

Specifies the number of linear segments in a quarter circle in the approximation of circular arcs.

`join_style : {'round', 'bevel', 'mitre'}, (optional, default 'round')`

Specifies the shape of outside corners. ‘round’ results in rounded shapes. ‘bevel’ results in a beveled edge that touches the original vertex. ‘mitre’ results in a single vertex that is beveled depending on the `mitre_limit` parameter.

`mitre_limit : float (optional, default 5.0)`

Crops off ‘mitre’-style joins if the point is displaced from the buffered vertex by more than this limit.

See [http://shapely.readthedocs.io/en/latest/manual.html#object.offset\\_curve](http://shapely.readthedocs.io/en/latest/manual.html#object.offset_curve) for details.

## Examples

```
>>> from shapely.geometry import LineString
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (0, 1), (1, 1)]),
...],
... crs=3857
...)
>>> s
0 LINESTRING (0 0, 0 1, 1 1)
dtype: geometry
```

```
>>> s.offset_curve(1)
0 LINESTRING (-1 0, -1 1, -0.981 1.195, -0.924 1...
dtype: geometry
```

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.hausdorff\_distance

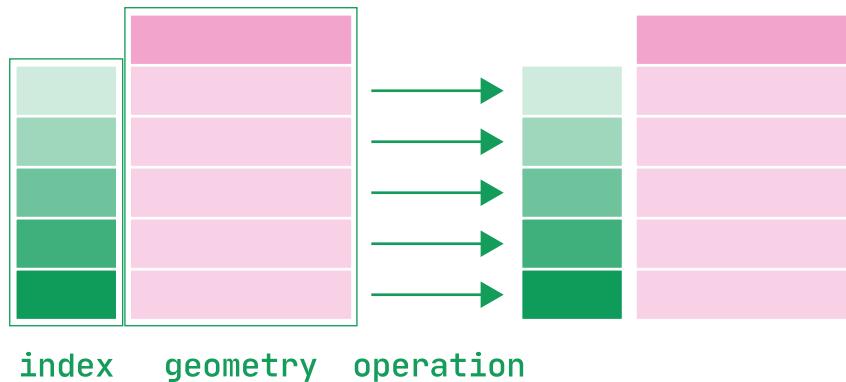
`GeoSeries.hausdorff_distance(other, align=None, densify=None)`

[\[source\]](#)

Returns a `Series` containing the Hausdorff distance to aligned `other`.

The Hausdorff distance is the largest distance consisting of any point in `self` with the nearest point in `other`.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The Geoseries (elementwise) or geometric object to find the distance to.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

`densify : float (default None)`

A value between 0 and 1, that splits each subsegment of a line string into equal length segments, making the approximation less coarse. A densify value of 0.5 will add a point halfway between each pair of points. A densify value of 0.25 will add a point a quarter of the way between each pair of points.

## Returns:

`Series (float)`

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 0), (1, 1)]),
... Polygon([(0, 0), (-1, 0), (-1, 1)]),
... LineString([(1, 1), (0, 0)]),
... Point(0, 0),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0.5, 0.5), (1.5, 0.5), (1.5, 1.5), (0.5, 1.5)]),
... Point(3, 1),
... LineString([(1, 0), (2, 0)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

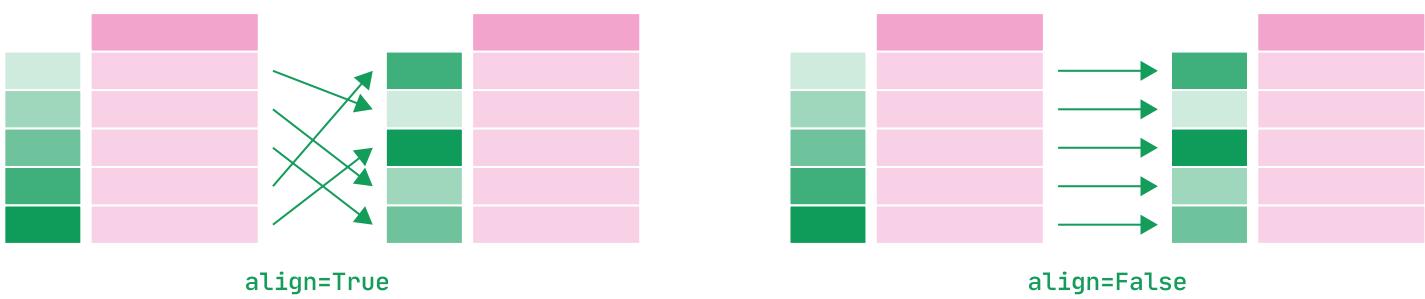
```
>>> s
0 POLYGON ((0 0, 1 0, 1 1, 0 0))
1 POLYGON ((0 0, -1 0, -1 1, 0 0))
2 LINESTRING (1 1, 0 0)
3 POINT (0 0)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0.5 0.5, 1.5 0.5, 1.5 1.5, 0.5 1.5, ...
2 POINT (3 1)
3 LINESTRING (1 0, 2 0)
4 POINT (0 1)
dtype: geometry
```

We can check the hausdorff distance of each geometry of GeoSeries to a single geometry:

```
>>> point = Point(-1, 0)
>>> s.hausdorff_distance(point)
0 2.236068
1 1.000000
2 2.236068
3 1.000000
dtype: float64
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and use elements with the same index using `align=True` or ignore index and use elements based on their matching order using `align=False`:



```
>>> s.hausdorff_distance(s2, align=True)
0 NaN
1 2.121320
2 3.162278
3 2.000000
4 NaN
dtype: float64
```

```
>>> s.hausdorff_distance(s2, align=False)
0 0.707107
1 4.123106
2 1.414214
3 1.000000
dtype: float64
```

We can also set a `densify` value, which is a float between 0 and 1 and signifies the fraction of the distance between each pair of points that will be used as the distance between the points when densifying.

# geopandas.GeoSeries.frechet\_distance

`GeoSeries.frechet_distance(other, align=None, densify=None)`

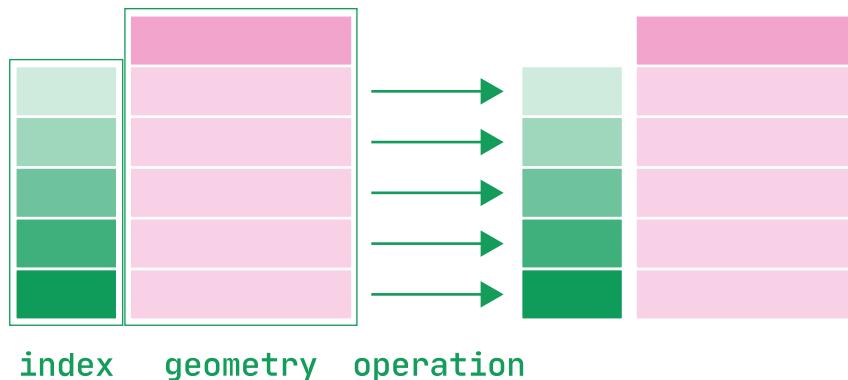
[\[source\]](#)

Returns a `Series` containing the Frechet distance to aligned `other`.

The Fréchet distance is a measure of similarity: it is the greatest distance between any point in A and the closest point in B. The discrete distance is an approximation of this metric: only vertices are considered. The parameter `densify` makes this approximation less coarse by splitting the line segments between vertices before computing the distance.

Fréchet distance sweep continuously along their respective curves and the direction of curves is significant. This makes it a better measure of similarity than Hausdorff distance for curve or surface matching.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The Geoseries (elementwise) or geometric object to find the distance to.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

`densify : float (default None)`

A value between 0 and 1, that splits each subsegment of a line string into equal length segments, making the approximation less coarse. A densify value of 0.5 will add a point halfway between each pair of points. A densify value of 0.25 will add a point every quarter of the way between each pair of points.

## Returns:

`Series (float)`

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 0), (1, 1)]),
... Polygon([(0, 0), (-1, 0), (-1, 1)]),
... LineString([(1, 1), (0, 0)]),
... Point(0, 0),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0.5, 0.5), (1.5, 0.5), (1.5, 1.5), (0.5, 1.5)]),
... Point(3, 1),
... LineString([(1, 0), (2, 0)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

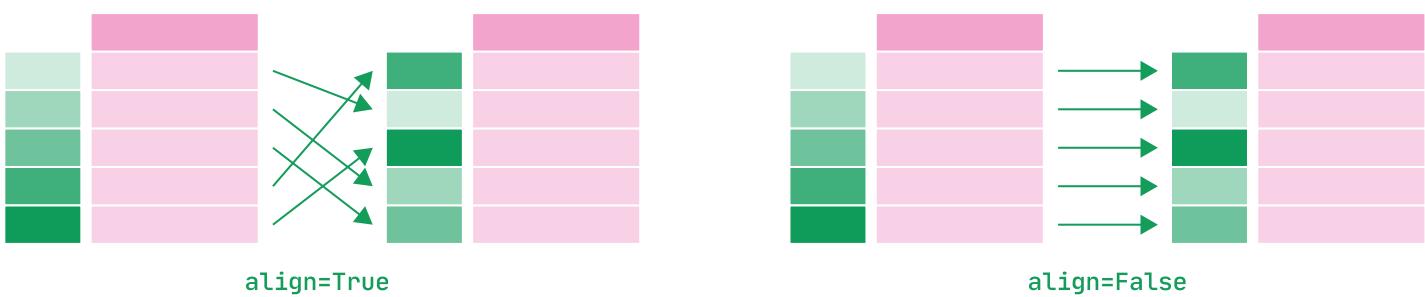
```
>>> s
0 POLYGON ((0 0, 1 0, 1 1, 0 0))
1 POLYGON ((0 0, -1 0, -1 1, 0 0))
2 LINESTRING (1 1, 0 0)
3 POINT (0 0)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0.5 0.5, 1.5 0.5, 1.5 1.5, 0.5 1.5, ...
2 POINT (3 1)
3 LINESTRING (1 0, 2 0)
4 POINT (0 1)
dtype: geometry
```

We can check the frechet distance of each geometry of GeoSeries to a single geometry:

```
>>> point = Point(-1, 0)
>>> s.frechet_distance(point)
0 2.236068
1 1.000000
2 2.236068
3 1.000000
dtype: float64
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and use elements with the same index using `align=True` or ignore index and use elements based on their matching order using `align=False`:



```
>>> s.frechet_distance(s2, align=True)
0 NaN
1 2.121320
2 3.162278
3 2.000000
4 NaN
dtype: float64
>>> s.frechet_distance(s2, align=False)
0 0.707107
1 4.123106
2 2.000000
3 1.000000
dtype: float64
```

We can also set a `densify` value, which is a float between 0 and 1 and signifies the fraction of the distance between each pair of points that will be used as the distance between the points when densifying.

```
>>> l1 = geopandas.GeoSeries([LineString([(0, 0), (10, 0), (0, 15)])])
>>> l2 = geopandas.GeoSeries([LineString([(0, 0), (20, 15), (9, 11)])])
>>> l1.frechet_distance(l2)
0 18.027756
dtype: float64
>>> l1.frechet_distance(l2, densify=0.25)
0 18.770511
```

# geopandas.GeoSeries.representative\_point

## GeoSeries.representative\_point()

[\[source\]](#)

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

### See also

[GeoSeries.centroid](#)

geometric centroid

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.representative_point()
0 POINT (0.25 0.5)
1 POINT (1 1)
2 POINT (0 0)
dtype: geometry
```

# geopandas.GeoSeries.exterior

*property* `GeoSeries.exterior`

[\[source\]](#)

Returns a `GeoSeries` of LinearRings representing the outer boundary of each polygon in the GeoSeries.

Applies to GeoSeries containing only Polygons. Returns `None`` for other geometry types.

## See also

[`GeoSeries.boundary`](#)

complete set-theoretic boundary

[`GeoSeries.interiors`](#)

list of inner rings of each polygon

## Examples

```
>>> from shapely.geometry import Polygon, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... Polygon([(1, 0), (2, 1), (0, 0)]),
... Point(0, 1)
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 POLYGON ((1 0, 2 1, 0 0, 1 0))
2 POINT (0 1)
dtype: geometry
```

```
>>> s.exterior
0 LINEARRING (0 0, 1 1, 0 1, 0 0)
1 LINEARRING (1 0, 2 1, 0 0, 1 0)
2 None
dtype: geometry
```

# geopandas.GeoSeries.interiors

*property* `GeoSeries.interiors`

[\[source\]](#)

Returns a `Series` of List representing the inner rings of each polygon in the GeoSeries.

Applies to GeoSeries containing only Polygons.

**Returns:**

`inner_rings: Series of List`

Inner rings of each polygon in the GeoSeries.

➡ See also

[`GeoSeries.exterior`](#)

outer boundary

## Examples

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [
... Polygon(
... [(0, 0), (0, 5), (5, 5), (5, 0)],
... [[(1, 1), (2, 1), (1, 2)], [(1, 4), (2, 4), (2, 3)]],
...),
... Polygon([(1, 0), (2, 1), (0, 0)])
...]
...)
>>> s
0 POLYGON ((0 0, 0 5, 5 5, 5 0, 0 0), (1 1, 2 1, ...
1 POLYGON ((1 0, 2 1, 0 0, 1 0))
dtype: geometry
```

```
>>> s.interiors
0 [LINEARRING (1 1, 2 1, 1 2, 1 1), LINEARRING (...)]
1
dtype: object
```

# geopandas.GeoSeries.minimum\_bounding\_

## GeoSeries.minimum\_bounding\_radius()

[\[source\]](#)

Returns a Series of the radii of the minimum bounding circles that enclose each geometry.

### See also

[GeoSeries.minimum\\_bounding\\_circle](#)

minimum bounding circle (geometry)

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0,0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.minimum_bounding_radius()
0 0.707107
1 0.707107
2 0.000000
dtype: float64
```

# geopandas.GeoSeries.x

*property* `GeoSeries.x`

[\[source\]](#)

Return the x location of point geometries in a GeoSeries

**Returns:**

`pandas.Series`

## See also

[GeoSeries.y](#)

[GeoSeries.z](#)

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)])
>>> s.x
0 1.0
1 2.0
2 3.0
dtype: float64
```



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.z

*property* `GeoSeries.z`

[\[source\]](#)

Return the z location of point geometries in a GeoSeries

**Returns:**

`pandas.Series`

## See also

[GeoSeries.x](#)

[GeoSeries.y](#)

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1, 1), Point(2, 2, 2), Point(3, 3, 3)])
>>> s.z
0 1.0
1 2.0
2 3.0
dtype: float64
```

# geopandas.GeoSeries.get\_coordinates

`GeoSeries.get_coordinates(include_z=False, ignore_index=False, index_parts=False)`

[\[source\]](#)

Gets coordinates from a `GeoSeries` as a `DataFrame` of floats.

The shape of the returned `DataFrame` is (N, 2), with N being the number of coordinate pairs. With the default of `include_z=False`, three-dimensional data is ignored. When specifying `include_z=True`, the shape of the returned `DataFrame` is (N, 3).

## Parameters:

`include_z : bool, default False`

Include Z coordinates

`ignore_index : bool, default False`

If True, the resulting index will be labelled 0, 1, ..., n - 1, ignoring `index_parts`.

`index_parts : bool, default False`

If True, the resulting index will be a `MultiIndex` (original index with an additional level indicating the ordering of the coordinate pairs: a new zero-based index for each geometry in the original GeoSeries).

## Returns:

`pandas.DataFrame`

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(1, 1),
... LineString([(1, -1), (1, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
>>> s
0 POINT (1 1)
1 LINESTRING (1 -1, 1 0)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.get_coordinates()
 x y
0 1.0 1.0
1 1.0 -1.0
1 1.0 0.0
2 3.0 -1.0
2 4.0 0.0
2 3.0 1.0
2 3.0 -1.0
```

```
>>> s.get_coordinates(ignore_index=True)
 x y
0 1.0 1.0
1 1.0 -1.0
2 1.0 0.0
3 3.0 -1.0
4 4.0 0.0
5 3.0 1.0
6 3.0 -1.0
```

```
>>> s.get_coordinates(index_parts=True)
 x y
0 0 1.0 1.0
1 0 1.0 -1.0
1 1 1.0 0.0
2 0 3.0 -1.0
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.count\_coordinates

## GeoSeries.count\_coordinates()

[\[source\]](#)

Returns a [Series](#) containing the count of the number of coordinate pairs in each geometry.

### See also

[GeoSeries.get\\_coordinates](#)

extract coordinates as a [DataFrame](#)

[GeoSeries.count\\_geometries](#)

count the number of geometries in a collection

## Examples

An example of a GeoDataFrame with two line strings, one point and one None value:

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (1, 1), (1, -1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, -1)]),
... Point(0, 0),
... Polygon([(10, 10), (10, 20), (20, 20), (20, 10), (10, 10)]),
... None
...]
...)
>>> s
0 LINESTRING (0 0, 1 1, 1 -1, 0 1)
1 LINESTRING (0 0, 1 1, 1 -1)
2 POINT (0 0)
3 POLYGON ((10 10, 10 20, 20 20, 20 10, 10 10))
4 None
dtype: geometry
```

```
>>> s.count_coordinates()
0 4
1 3
2 1
3 5
4 0
dtype: int32
```

# geopandas.GeoSeries.count\_geometries

`GeoSeries.count_geometries()`

[\[source\]](#)

Returns a `Series` containing the count of geometries in each multi-part geometry.

For single-part geometry objects, this is always 1. For multi-part geometries, like `MultiPoint` or `MultiLineString`, it is the number of parts in the geometry. For `GeometryCollection`, it is the number of geometries direct parts of the collection (the method does not recurse into collections within collections).

## See also

[`GeoSeries.count\_coordinates`](#)

count the number of coordinates in a geometry

[`GeoSeries.count\_interior\_rings`](#)

count the number of interior rings

## Examples

```
>>> from shapely.geometry import Point, MultiPoint, LineString, MultiLineString >>>
>>> s = geopandas.GeoSeries(
... [
... MultiPoint([(0, 0), (1, 1), (1, -1), (0, 1)]),
... MultiLineString([(0, 0), (1, 1), (-1, 0), (1, 0)]),
... LineString([(0, 0), (1, 1), (1, -1)]),
... Point(0, 0),
...]
...)
>>> s
0 MULTIPOLY ((0 0), (1 1), (1 -1), (0 1))
1 MULTILINESTRING ((0 0, 1 1), (-1 0, 1 0))
2 LINESTRING (0 0, 1 1, 1 -1)
3 POINT (0 0)
dtype: geometry
```

```
>>> s.count_geometries() >>>
0 4
1 2
2 1
3 1
dtype: int32
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.count\_interior\_rings

`GeoSeries.count_interior_rings()`

[\[source\]](#)

Returns a `Series` containing the count of the number of interior rings in a polygonal geometry.

For non-polygonal geometries, this is always 0.

## See also

[`GeoSeries.count\_coordinates`](#)

count the number of coordinates in a geometry

[`GeoSeries.count\_geometries`](#)

count the number of geometries in a collection

## Examples

```
>>> from shapely.geometry import Polygon, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon(
... [((0, 0), (0, 5), (5, 5), (5, 0)),
... [((1, 1), (1, 4), (4, 4), (4, 1))]),
...),
... Polygon(
... [((0, 0), (0, 5), (5, 5), (5, 0)),
... [
... [((1, 1), (1, 2), (2, 2), (2, 1)),
... [((3, 2), (3, 3), (4, 3), (4, 2))],
...],
...),
... Point(0, 1),
...]
...)
...)
>>> s
0 POLYGON ((0 0, 0 5, 5 5, 5 0, 0 0), (1 1, 1 4, ...
1 POLYGON ((0 0, 0 5, 5 5, 5 0, 0 0), (1 1, 1 2, ...
2 POINT (0 1)
dtype: geometry
```

```
>>> s.count_interior_rings()
0 1
1 2
2 0
dtype: int32
```



# geopandas.GeoSeries.set\_precision

`GeoSeries.set_precision(grid_size, mode='valid_output')`

[\[source\]](#)

Returns a `GeoSeries` with the precision set to a precision grid size.

By default, geometries use double precision coordinates (`grid_size=0`).

Coordinates will be rounded if a precision grid is less precise than the input geometry. Duplicated vertices will be dropped from lines and polygons for grid sizes greater than 0. Line and polygon geometries may collapse to empty geometries if all vertices are closer together than `grid_size`. Spikes or sections in Polygons narrower than `grid_size` after rounding the vertices will be removed, which can lead to MultiPolygons or empty geometries. Z values, if present, will not be modified.

## Parameters:

`grid_size : float`

Precision grid size. If 0, will use double precision (will not modify geometry if precision grid size was not previously set). If this value is more precise than input geometry, the input geometry will not be modified.

`mode : {'valid_output', 'pointwise', 'keep_collapsed'}, default 'valid_output'`

This parameter determines the way a precision reduction is applied on the geometry. There are three modes:

- `'valid_output'` (default): The output is always valid. Collapsed geometry elements (including both polygons and lines) are removed. Duplicate vertices are removed.
- `'pointwise'`: Precision reduction is performed pointwise. Output geometry may be invalid due to collapse or self-intersection. Duplicate vertices are not removed.
- `'keep_collapsed'`: Like the default mode, except that collapsed linear geometry elements are preserved. Collapsed polygonal input elements are removed. Duplicate vertices are removed.

## Notes

Subsequent operations will always be performed in the precision of the geometry with higher precision (smaller `grid_size`). That same precision will be attached to the operation outputs.

Input geometries should be geometrically valid; unexpected results may occur if input geometries are not. You can check the validity with `is_valid()` and fix invalid geometries with `make_valid()` methods.

## Examples

```
>>> from shapely import LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Point(0.9, 0.9),
... Point(0.9, 0.9, 0.9),
... LineString([(0, 0), (0, 0.1), (0, 1), (1, 1)]),
... LineString([(0, 0), (0, 0.1), (0.1, 0.1)])
...],
...)
>>> s
0 POINT (0.9 0.9)
1 POINT Z (0.9 0.9 0.9)
2 LINESTRING (0 0, 0 0.1, 0 1, 1 1)
3 LINESTRING (0 0, 0 0.1, 0.1 0.1)
dtype: geometry
```

```
>>> s.set_precision(1)
0 POINT (1 1)
1 POINT Z (1 1 0.9)
2 LINESTRING (0 0, 0 1, 1 1)
3 LINESTRING Z EMPTY
dtype: geometry
```

```
>>> s.set_precision(1, mode="pointwise")
0 POINT (1 1)
1 POINT Z (1 1 0.9)
2 LINESTRING (0 0, 0 0, 0 1, 1 1)
3 LINESTRING (0 0, 0 0, 0 0)
dtype: geometry
```

```
>>> s.set_precision(1, mode="keepCollapsed")
0 POINT (1 1)
1 POINT Z (1 1 0.9)
2 LINESTRING (0 0, 0 1, 1 1)
3 LINESTRING (0 0, 0 0)
dtype: geometry
```



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.get\_geometry

`GeoSeries.get_geometry(index)`

[\[source\]](#)

Returns the n-th geometry from a collection of geometries.

## Parameters:

`index : int or array_like`

Position of a geometry to be retrieved within its collection

## Returns:

`GeoSeries`

## Notes

Simple geometries act as collections of length 1. Any out-of-range index value returns None.

## Examples

```
>>> from shapely.geometry import Point, MultiPoint, GeometryCollection
>>> s = geopandas.GeoSeries(
... [
... Point(0, 0),
... MultiPoint([(0, 0), (1, 1), (0, 1), (1, 0)]),
... GeometryCollection(
... [MultiPoint([(0, 0), (1, 1), (0, 1), (1, 0)]), Point(0, 1)]
...),
...]
...)
>>> s
0 POINT (0 0)
1 MULTIPOLY ((0 0), (1 1), (0 1), (1 0))
2 GEOMETRYCOLLECTION (MULTIPOINT ((0 0), (1 1), ...
dtype: geometry
```

```
>>> s.get_geometry(0)
0 POINT (0 0)
1 POINT (0 0)
2 MULTIPOLY ((0 0), (1 1), (0 1), (1 0))
dtype: geometry
```

```
>>> s.get_geometry(1)
0 None
1 POINT (1 1)
2 POINT (0 1)
dtype: geometry
```

```
>>> s.get_geometry(-1)
0 POINT (0 0)
1 POINT (1 0)
2 POINT (0 1)
dtype: geometry
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.is\_closed

property `GeoSeries.is_closed`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` if a LineString's or LinearRing's first and last points are equal.

Returns False for any other geometry type.

## Examples

```
>>> from shapely.geometry import LineString, Point, Polygon
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(0, 0), (1, 1), (0, 1)]),
... Polygon([(0, 0), (0, 1), (1, 1), (0, 0)]),
... Point(3, 3)
...]
...)
>>> s
0 LINESTRING (0 0, 1 1, 0 1, 0 0)
1 LINESTRING (0 0, 1 1, 0 1)
2 POLYGON ((0 0, 0 1, 1 1, 0 0))
3 POINT (3 3)
dtype: geometry
```

```
>>> s.is_closed
0 True
1 False
2 False
3 False
dtype: bool
```

# geopandas.GeoSeries.is\_empty

property `GeoSeries.is_empty`

[\[source\]](#)

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

## See also

[`GeoSeries.isna`](#)

detect missing values

## Examples

An example of a GeoDataFrame with one empty point, one point and one missing value:

```
>>> from shapely.geometry import Point
>>> d = {'geometry': [Point(), Point(2, 1), None]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf
 geometry
0 POINT EMPTY
1 POINT (2 1)
2 None
```

```
>>> gdf.is_empty
0 True
1 False
2 False
dtype: bool
```

# geopandas.GeoSeries.is\_ring

property `GeoSeries.is_ring`

[\[source\]](#)

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

When constructing a LinearRing, the sequence of coordinates may be explicitly closed by passing identical values in the first and last indices. Otherwise, the sequence will be implicitly closed by copying the first tuple to the last index.

## Examples

```
>>> from shapely.geometry import LineString, LinearRing
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (1, 1), (1, -1)]),
... LineString([(0, 0), (1, 1), (1, -1), (0, 0)]),
... LinearRing([(0, 0), (1, 1), (1, -1)]),
...]
...)
>>> s
0 LINESTRING (0 0, 1 1, 1 -1)
1 LINESTRING (0 0, 1 1, 1 -1, 0 0)
2 LINEARRING (0 0, 1 1, 1 -1, 0 0)
dtype: geometry
```

```
>>> s.is_ring
0 False
1 True
2 True
dtype: bool
```

# geopandas.GeoSeries.is\_simple

*property* `GeoSeries.is_simple`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for geometries that do not cross themselves.

This is meaningful only for *LineStrings* and *LinearRings*.

## Examples

```
>>> from shapely.geometry import LineString
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (1, 1), (1, -1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, -1)]),
...]
...)
>>> s
0 LINESTRING (0 0, 1 1, 1 -1, 0 1)
1 LINESTRING (0 0, 1 1, 1 -1)
dtype: geometry
```

```
>>> s.is_simple
0 False
1 True
dtype: bool
```

# geopandas.GeoSeries.is\_valid

property `GeoSeries.is_valid`

[\[source\]](#)

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

## See also

[`GeoSeries.is\_valid\_reason`](#)

reason for invalidity

## Examples

An example with one invalid polygon (a bowtie geometry crossing itself) and one missing geometry:

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... Polygon([(0,0), (1, 1), (1, 0), (0, 1)]), # bowtie geometry
... Polygon([(0, 0), (2, 2), (2, 0)]),
... None
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 POLYGON ((0 0, 1 1, 1 0, 0 1, 0 0))
2 POLYGON ((0 0, 2 2, 2 0, 0 0))
3 None
dtype: geometry
```

```
>>> s.is_valid
0 True
1 False
2 True
3 False
dtype: bool
```

# geopandas.GeoSeries.is\_valid\_reason

## GeoSeries.is\_valid\_reason()

[\[source\]](#)

Returns a `Series` of strings with the reason for invalidity of each geometry.

### See also

[`GeoSeries.is\_valid`](#)

detect invalid geometries

[`GeoSeries.make\_valid`](#)

fix invalid geometries

## Examples

An example with one invalid polygon (a bowtie geometry crossing itself) and one missing geometry:

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... Polygon([(0,0), (1, 1), (1, 0), (0, 1)]), # bowtie geometry
... Polygon([(0, 0), (2, 2), (2, 0)]),
... None
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 POLYGON ((0 0, 1 1, 1 0, 0 1, 0 0))
2 POLYGON ((0 0, 2 2, 2 0, 0 0))
3 None
dtype: geometry
```

```
>>> s.is_valid_reason()
0 Valid Geometry
1 Self-intersection[0.5 0.5]
2 Valid Geometry
3 None
dtype: object
```

# geopandas.GeoSeries.has\_z

property `GeoSeries.has_z`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for features that have a z-component.

## Notes

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries(
... [
... Point(0, 1),
... Point(0, 1, 2),
...]
...)
>>> s
0 POINT (0 1)
1 POINT Z (0 1 2)
dtype: geometry
```

```
>>> s.has_z
0 False
1 True
dtype: bool
```

# geopandas.GeoSeries.is\_ccw

*property* `GeoSeries.is_ccw`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` if a LineString or LinearRing is counter-clockwise.

Note that there are no checks on whether lines are actually closed and not self-intersecting, while this is a requirement for `is_ccw`. The recommended usage of this property for LineStrings is `GeoSeries.is_ccw & GeoSeries.is_simple` and for LinearRings `GeoSeries.is_ccw & GeoSeries.is_valid`.

This property will return False for non-linear geometries and for lines with fewer than 4 points (including the closing point).

## Examples

```
>>> from shapely.geometry import LineString, LinearRing, Point
>>> s = geopandas.GeoSeries(
... [
... LinearRing([(0, 0), (0, 1), (1, 1), (0, 0)]),
... LinearRing([(0, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(0, 0), (1, 1), (0, 1)]),
... Point(3, 3)
...]
...)
>>> s
0 LINEARRING (0 0, 0 1, 1 1, 0 0)
1 LINEARRING (0 0, 1 1, 0 1, 0 0)
2 LINESTRING (0 0, 1 1, 0 1)
3 POINT (3 3)
dtype: geometry
```

```
>>> s.is_ccw
0 False
1 True
2 False
3 False
dtype: bool
```

# geopandas.GeoSeries.contains

`GeoSeries.contains(other, align=None)`

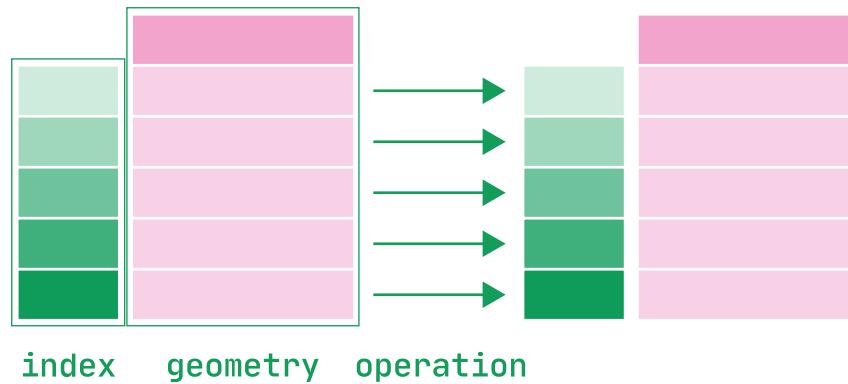
[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that contains `other`.

An object is said to contain `other` if at least one point of `other` lies in the interior and no points of `other` lie in the exterior of the object. (Therefore, any given polygon does not contain its own boundary - there is not any point that lies in the interior.) If either object is empty, this operation returns `False`.

This is the inverse of [`within\(\)`](#) in the sense that the expression `a.contains(b) == b.within(a)` always evaluates to `True`.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if it is contained.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.contains\_properly`](#)

[`GeoSeries.within`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries `contains` any element of the other one.

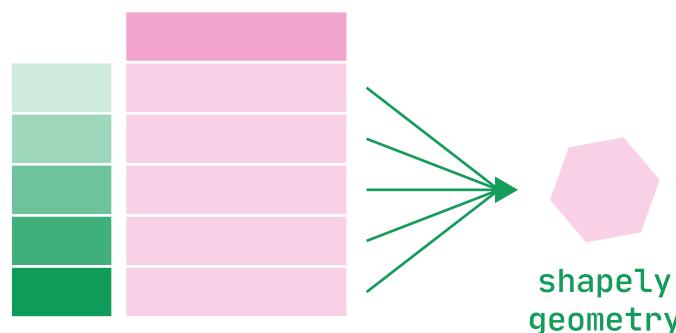
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (0, 2)]),
... LineString([(0, 0), (0, 1)]),
... Point(0, 1),
...],
... index=range(0, 4),
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (1, 2), (0, 2)]),
... LineString([(0, 0), (0, 2)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 0 2)
2 LINESTRING (0 0, 0 1)
3 POINT (0 1)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 POLYGON ((0 0, 1 2, 0 2, 0 0))
3 LINESTRING (0 0, 0 2)
4 POINT (0 1)
dtype: geometry
```

We can check if each geometry of GeoSeries contains a single geometry:



```
>>> point = Point(0, 1)
>>> s.contains(point)
0 False
1 True
2 False
3 True
dtype: bool
```

>>>

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s2.contains(s, align=True)
0 False
1 False
2 False
3 True
4 False
dtype: bool
```

>>>

```
>>> s2.contains(s, align=False)
1 True
2 False
3 True
4 False
```

>>>

# geopandas.GeoSeries.contains\_properly

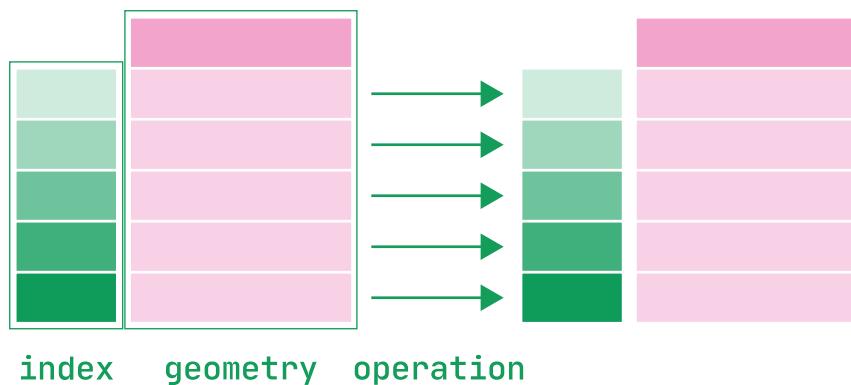
`GeoSeries.contains_properly(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is completely inside `other`, with no common boundary points.

Geometry A contains geometry B properly if B intersects the interior of A but not the boundary (or exterior). This means that a geometry A does not “contain properly” itself, which contrasts with the [`contains\(\)`](#) method, where common points on the boundary are allowed.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if it is contained.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.contains`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries `contains_properly` any element of the other one.

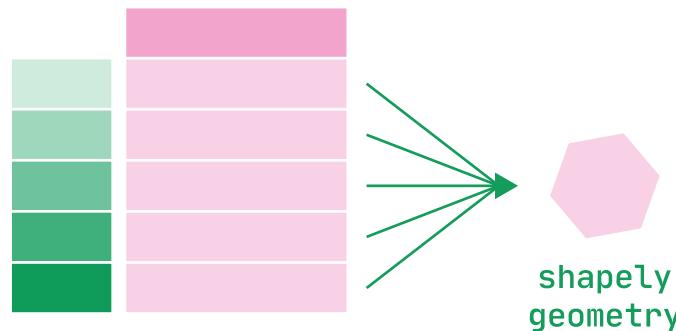
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (0, 2)]),
... LineString([(0, 0), (0, 1)]),
... Point(0, 1),
...],
... index=range(0, 4),
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (1, 2), (0, 2)]),
... LineString([(0, 0), (0, 2)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 0 2)
2 LINESTRING (0 0, 0 1)
3 POINT (0 1)
dtype: geometry
```

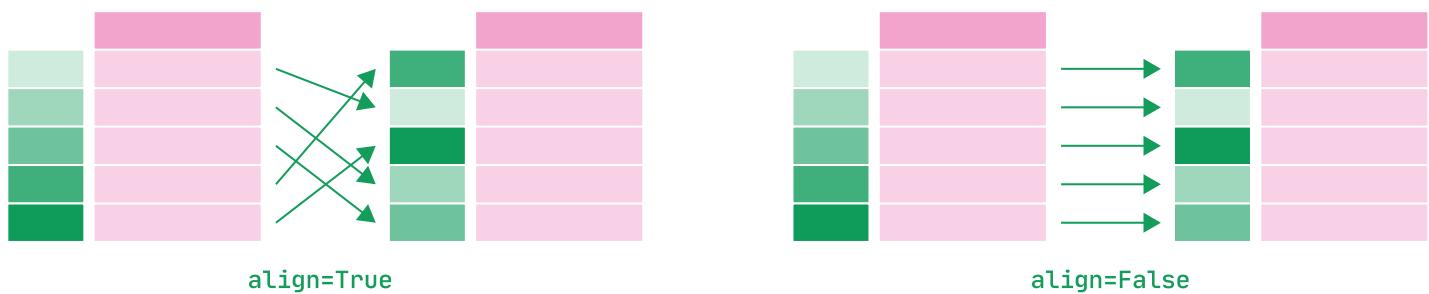
```
>>> s2
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 POLYGON ((0 0, 1 2, 0 2, 0 0))
3 LINESTRING (0 0, 0 2)
4 POINT (0 1)
dtype: geometry
```

We can check if each geometry of GeoSeries contains a single geometry:



```
>>> point = Point(0, 1)
>>> s.contains_properly(point)
0 False
1 True
2 False
3 True
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s2.contains_properly(s, align=True)
0 False
1 False
2 False
3 True
4 False
dtype: bool
```

```
>>> s2.contains_properly(s, align=False)
1 False
2 False
3 False
4 True
dtype: bool
```

Compare it to the result of [`contains\(\)`](#):

```
>>> s2.contains(s, align=False)
1 True
2 False
3 True
4 True
dtype: bool
```

# geopandas.GeoSeries.crosses

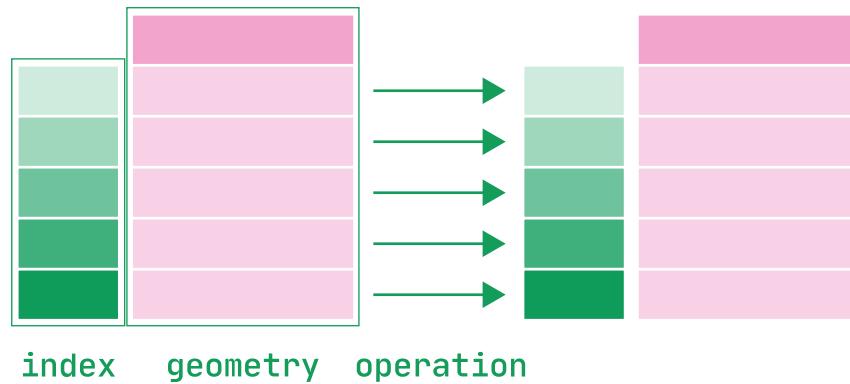
`GeoSeries.crosses(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that cross `other`.

An object is said to cross `other` if its *interior* 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.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if is crossed.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.disjoint`](#)

[`GeoSeries.intersects`](#)

## Notes

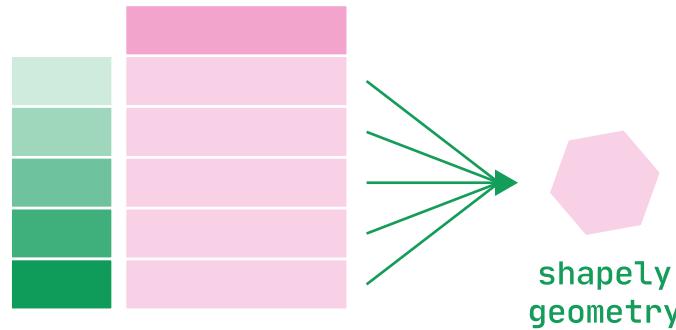
This method works in a row-wise manner. It does not check if an element of one GeoSeries `crosses` any element of the other one.

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 LINESTRING (0 0, 2 2)
2 LINESTRING (2 0, 0 2)
3 POINT (0 1)
dtype: geometry
>>> s2
1 LINESTRING (1 0, 1 3)
2 LINESTRING (2 0, 0 2)
3 POINT (1 1)
4 POINT (0 1)
dtype: geometry
```

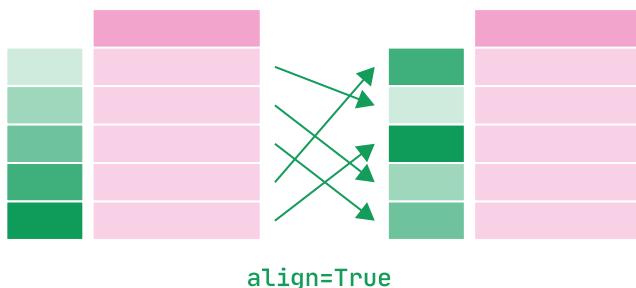
We can check if each geometry of GeoSeries crosses a single geometry:



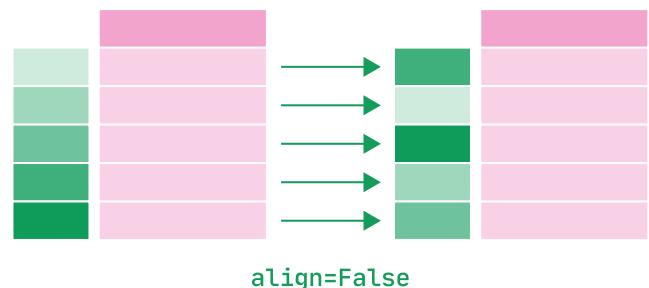
```
>>> line = LineString([(-1, 1), (3, 1)])
>>> s.crosses(line)
0 True
1 True
2 True
3 False
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the

same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



`align=True`



`align=False`

```
>>> s.crosses(s2, align=True)
0 False
1 True
2 False
3 False
4 False
dtype: bool
```

```
>>> s.crosses(s2, align=False)
0 True
1 True
2 False
3 False
dtype: bool
```

Notice that a line does not cross a point that it contains.

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.disjoint

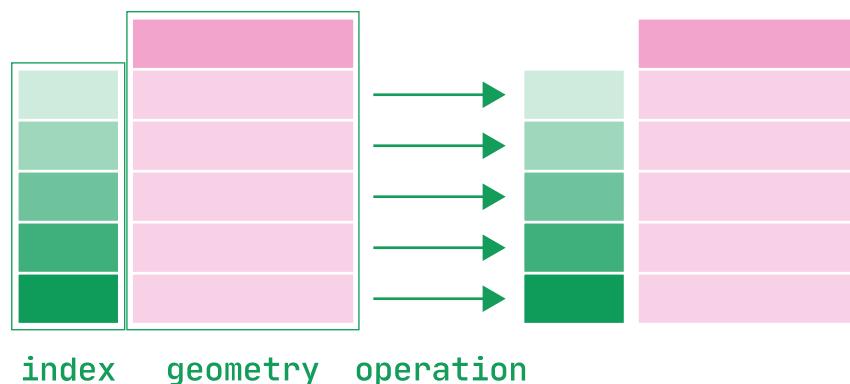
`GeoSeries.disjoint(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry disjoint to `other`.

An object is said to be disjoint to `other` if its *boundary* and *interior* does not intersect at all with those of the other.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if is disjoint.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.intersects`](#)

[`GeoSeries.touches`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is equal to any element of the other one.

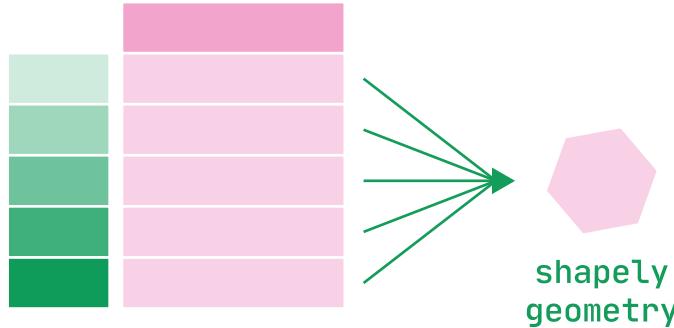
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(-1, 0), (-1, 2), (0, -2)]),
... LineString([(0, 0), (0, 1)]),
... Point(1, 1),
... Point(0, 0),
...],
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 LINESTRING (0 0, 2 2)
2 LINESTRING (2 0, 0 2)
3 POINT (0 1)
dtype: geometry
```

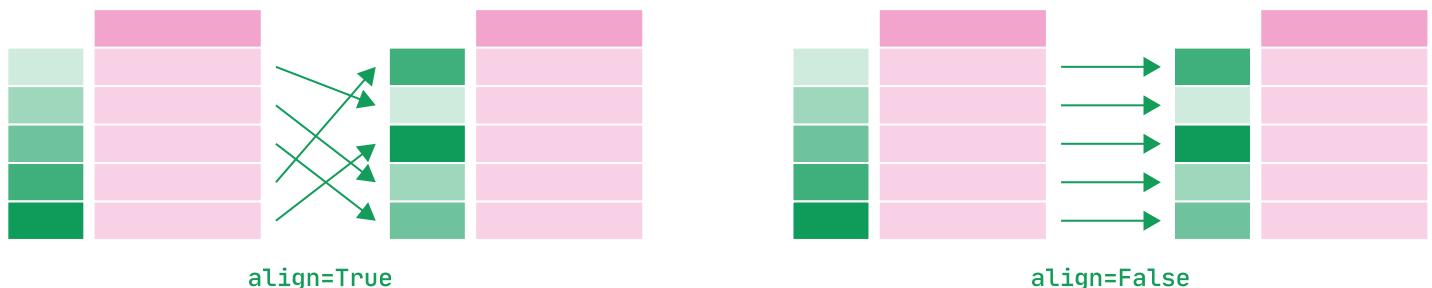
```
>>> s2
0 POLYGON ((-1 0, -1 2, 0 -2, -1 0))
1 LINESTRING (0 0, 0 1)
2 POINT (1 1)
3 POINT (0 0)
dtype: geometry
```

We can check each geometry of GeoSeries to a single geometry:



```
>>> line = LineString([(0, 0), (2, 0)])
>>> s.disjoint(line)
0 False
1 False
2 False
3 True
dtype: bool
```

We can also check two GeoSeries against each other, row by row. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.disjoint(s2)
0 True
1 False
2 False
3 True
dtype: bool
```

>>>

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

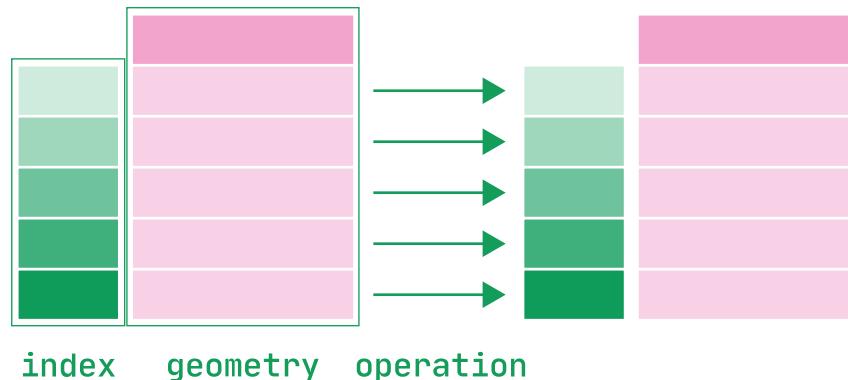
# geopandas.GeoSeries.dwithin

`GeoSeries.dwithin(other, distance, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is within a set distance from `other`.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test for equality.

`distance : float, np.array, pd.Series`

Distance(s) to test if each geometry is within. A scalar distance will be applied to all geometries. An array or Series will be applied elementwise. If np.array or pd.Series are used then it must have same length as the GeoSeries.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.within`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is within the set distance of *any* element of the other one.

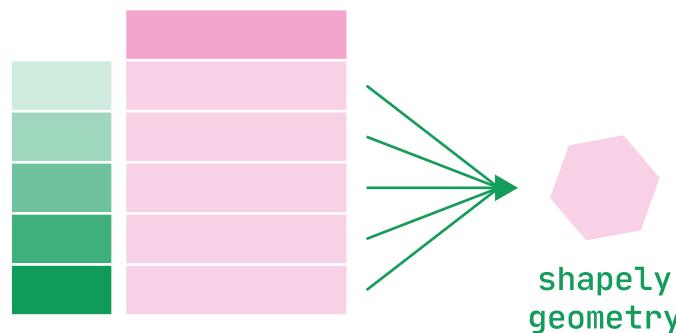
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (0, 2)]),
... LineString([(0, 0), (0, 1)]),
... Point(0, 1),
...],
... index=range(0, 4),
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(1, 0), (4, 2), (2, 2)]),
... Polygon([(2, 0), (3, 2), (2, 2)]),
... LineString([(2, 0), (2, 2)]),
... Point(1, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 0 2)
2 LINESTRING (0 0, 0 1)
3 POINT (0 1)
dtype: geometry
```

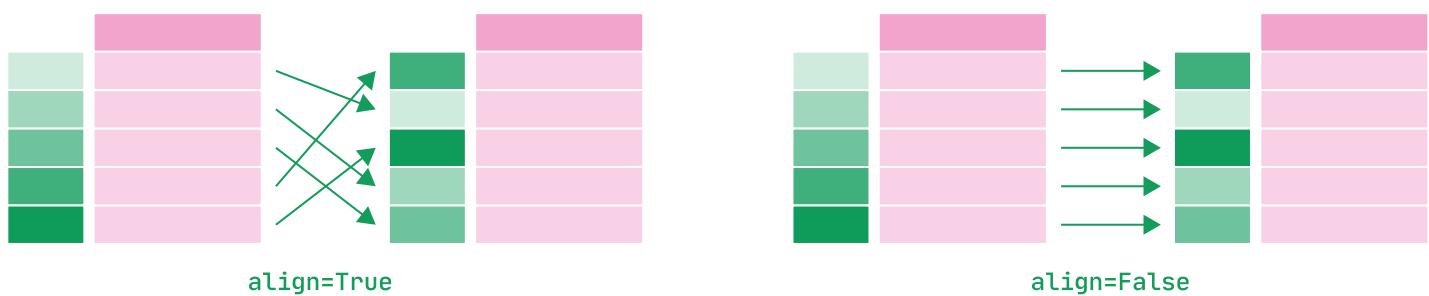
```
>>> s2
1 POLYGON ((1 0, 4 2, 2 2, 1 0))
2 POLYGON ((2 0, 3 2, 2 2, 2 0))
3 LINESTRING (2 0, 2 2)
4 POINT (1 1)
dtype: geometry
```

We can check if each geometry of GeoSeries contains a single geometry:



```
>>> point = Point(0, 1)
>>> s2.dwithin(point, 1.8)
1 True
2 False
3 False
4 True
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.dwithin(s2, distance=1, align=True)
0 False
1 True
2 False
3 False
4 False
dtype: bool
```

```
>>> s.dwithin(s2, distance=1, align=False)
0 True
1 False
2 False
3 True
dtype: bool
```

# geopandas.GeoSeries.geom\_equals

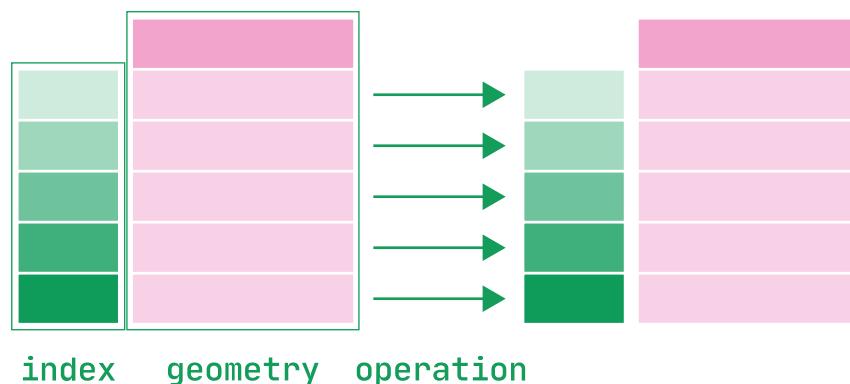
`GeoSeries.geom_equals(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry equal to `other`.

An object is said to be equal to `other` if its set-theoretic *boundary*, *interior*, and *exterior* coincides with those of the other.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test for equality.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.geom\_equals\_exact`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is equal to any element of the other one.

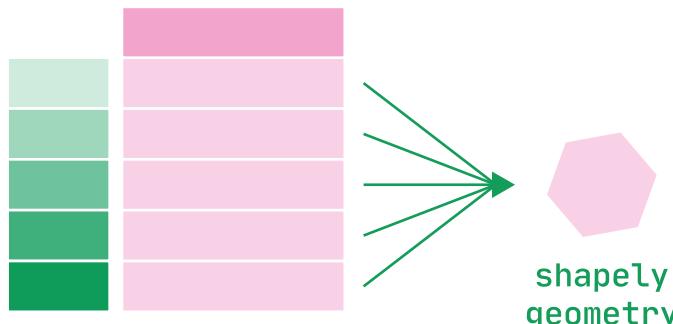
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (1, 2), (0, 2)]),
... LineString([(0, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (1, 2), (0, 2)]),
... Point(0, 1),
... LineString([(0, 0), (0, 2)]),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 1 2, 0 2, 0 0))
2 LINESTRING (0 0, 0 2)
3 POINT (0 1)
dtype: geometry
```

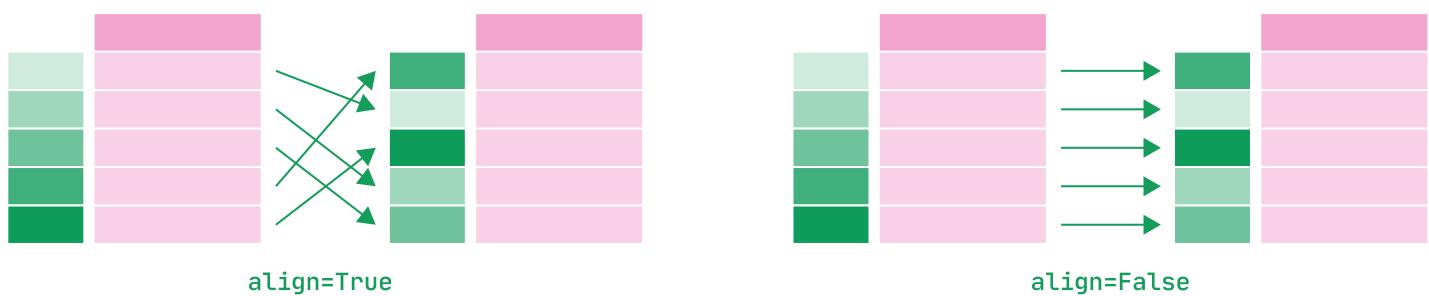
```
>>> s2
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 POLYGON ((0 0, 1 2, 0 2, 0 0))
3 POINT (0 1)
4 LINESTRING (0 0, 0 2)
dtype: geometry
```

We can check if each geometry of GeoSeries contains a single geometry:



```
>>> polygon = Polygon([(0, 0), (2, 2), (0, 2)])
>>> s.geom_equals(polygon)
0 True
1 False
2 False
3 False
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.geom_equals(s2)
0 False
1 False
2 False
3 True
4 False
dtype: bool
```

```
>>> s.geom_equals(s2, align=False)
0 True
1 True
2 False
3 False
dtype: bool
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.geom\_almost\_equals

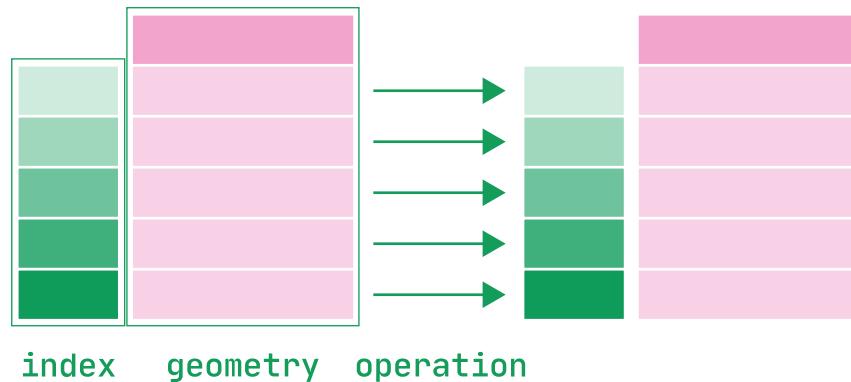
`GeoSeries.geom_almost_equals(other, decimal=6, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` if each aligned geometry is approximately equal to `other`.

Approximate equality is tested at all points to the specified `decimal` place precision.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to compare to.

`decimal : int`

Decimal place precision used when testing for approximate equality.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.geom\_equals`](#)

[`GeoSeries.geom\_equals\_exact`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is equal to any element of the other one.

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries(
... [
... Point(0, 1.1),
... Point(0, 1.01),
... Point(0, 1.001),
...],
...)
>>> s
0 POINT (0 1.1)
1 POINT (0 1.01)
2 POINT (0 1.001)
dtype: geometry
```

```
>>> s.geom_almost_equals(Point(0, 1), decimal=2)
0 False
1 False
2 True
dtype: bool
```

```
>>> s.geom_almost_equals(Point(0, 1), decimal=1)
0 False
1 True
2 True
dtype: bool
```

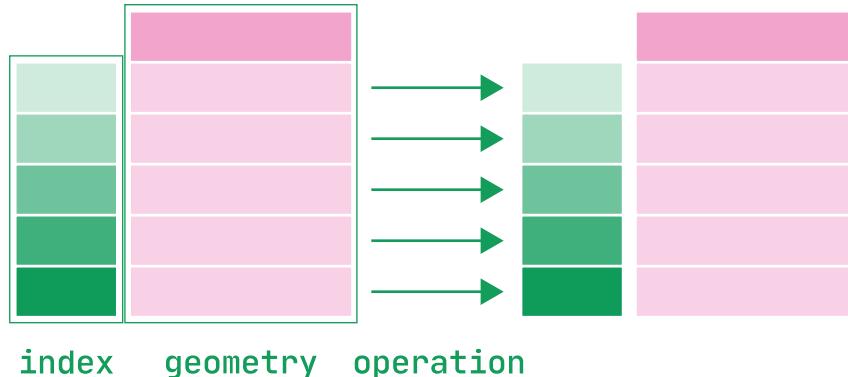
# geopandas.GeoSeries.geom\_equals\_exact

`GeoSeries.geom_equals_exact(other, tolerance, align=None)`

[\[source\]](#)

Return True for all geometries that equal aligned *other* to a given tolerance, else False.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

**other : *GeoSeries or geometric object***

The GeoSeries (elementwise) or geometric object to compare to.

**tolerance : *float***

Decimal place precision used when testing for approximate equality.

**align : *bool | None (default None)***

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.geom\_equals`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is equal to any element of the other one.

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries(
... [
... Point(0, 1.1),
... Point(0, 1.0),
... Point(0, 1.2),
...]
...)
>>> s
0 POINT (0 1.1)
1 POINT (0 1)
2 POINT (0 1.2)
dtype: geometry
```

```
>>> s.geom_equals_exact(Point(0, 1), tolerance=0.1)
0 False
1 True
2 False
dtype: bool
```

```
>>> s.geom_equals_exact(Point(0, 1), tolerance=0.15)
0 True
1 True
2 False
dtype: bool
```

# geopandas.GeoSeries.intersects

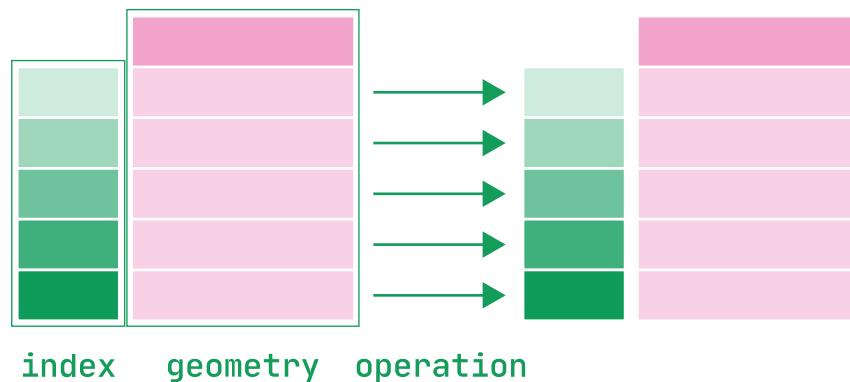
`GeoSeries.intersects(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that intersects `other`.

An object is said to intersect `other` if its *boundary* and *interior* intersects in any way with those of the other.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if is intersected.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.disjoint`](#)  
[`GeoSeries.crosses`](#)  
[`GeoSeries.touches`](#)  
[`GeoSeries.intersection`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries `crosses` any element of the other one.

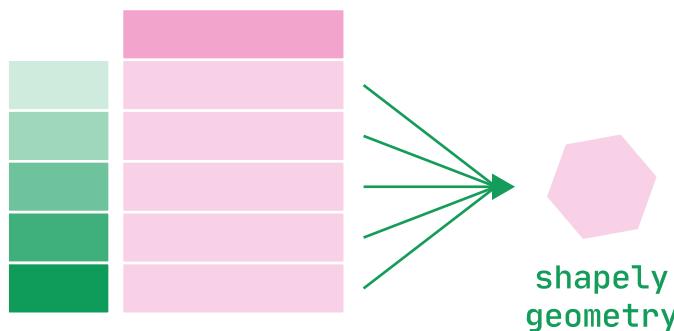
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 LINESTRING (0 0, 2 2)
2 LINESTRING (2 0, 0 2)
3 POINT (0 1)
dtype: geometry
```

```
>>> s2
1 LINESTRING (1 0, 1 3)
2 LINESTRING (2 0, 0 2)
3 POINT (1 1)
4 POINT (0 1)
dtype: geometry
```

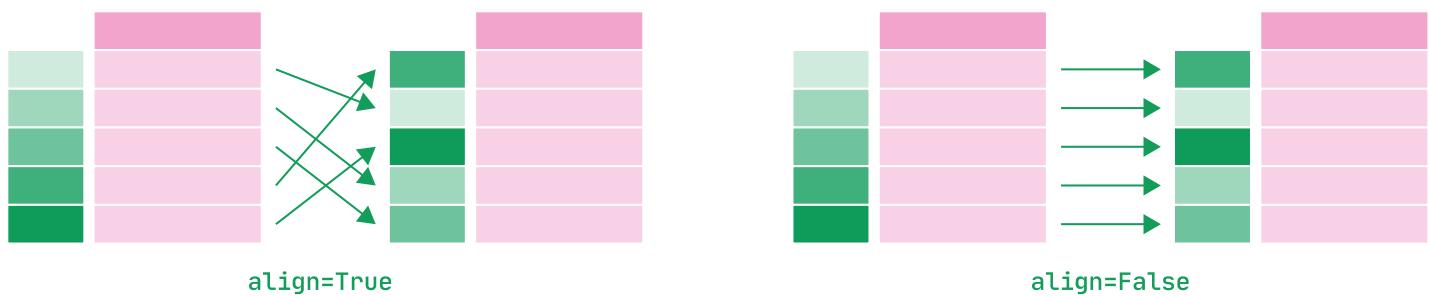
We can check if each geometry of GeoSeries crosses a single geometry:



```
>>> line = LineString([(-1, 1), (3, 1)])
>>> s.intersects(line)
0 True
1 True
2 True
3 True
dtype: bool
```

>>>

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.intersects(s2, align=True)
0 False
1 True
2 True
3 False
4 False
dtype: bool
```

>>>

```
>>> s.intersects(s2, align=False)
0 True
1 -
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.overlaps

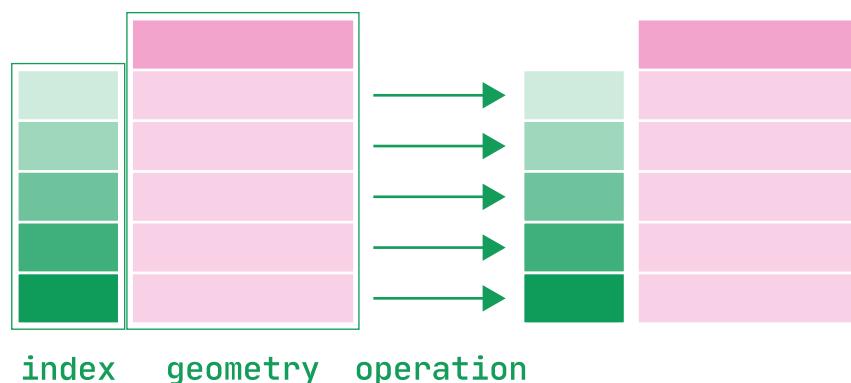
`GeoSeries.overlaps(other, align=None)`

[\[source\]](#)

Returns True for all aligned geometries that overlap *other*, else False.

Geometries overlaps if they have more than one but not all points in common, have the same dimension, and the intersection of the interiors of the geometries has the same dimension as the geometries themselves.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if overlaps.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.crosses`](#)

[`GeoSeries.intersects`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries `overlaps` any element of the other one.

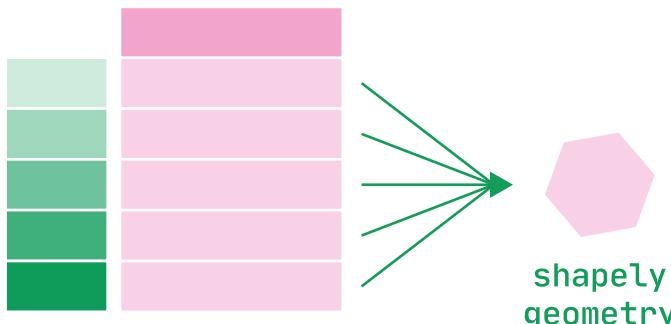
## Examples

```
>>> from shapely.geometry import Polygon, LineString, MultiPoint, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... MultiPoint([(0, 0), (0, 1)]),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 0), (0, 2), (0, 0)]),
... LineString([(0, 1), (1, 1)]),
... LineString([(1, 1), (3, 3)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 MULTIPOINT ((0 0), (0 1))
dtype: geometry
```

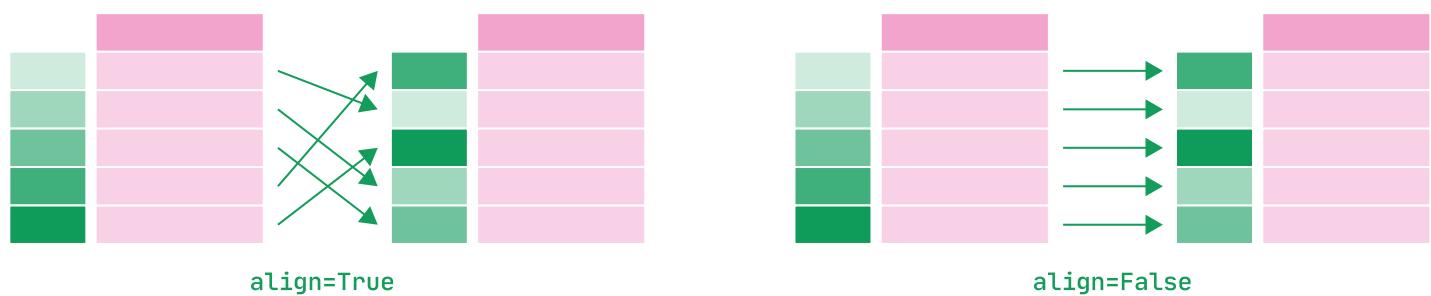
```
>>> s2
1 POLYGON ((0 0, 2 0, 0 2, 0 0))
2 LINESTRING (0 1, 1 1)
3 LINESTRING (1 1, 3 3)
4 POINT (0 1)
dtype: geometry
```

We can check if each geometry of GeoSeries overlaps a single geometry:



```
>>> polygon = Polygon([(0, 0), (1, 0), (1, 1), (0, 1)])
>>> s.overlaps(polygon)
0 True
1 True
2 False
3 False
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.overlaps(s2)
0 False
1 True
2 False
3 False
4 False
dtype: bool
```

```
>>> s.overlaps(s2, align=False)
0 True
1 False
2 True
3 False
dtype: bool
```

# geopandas.GeoSeries.touches

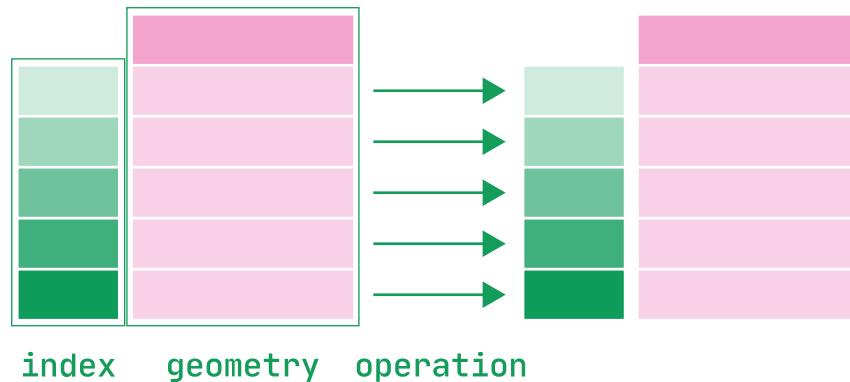
`GeoSeries.touches(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that touches `other`.

An object is said to touch `other` if it has at least one point in common with `other` and its interior does not intersect with any part of the other. Overlapping features therefore do not touch.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if is touched.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.overlaps`](#)

[`GeoSeries.intersects`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries `touches` any element of the other one.

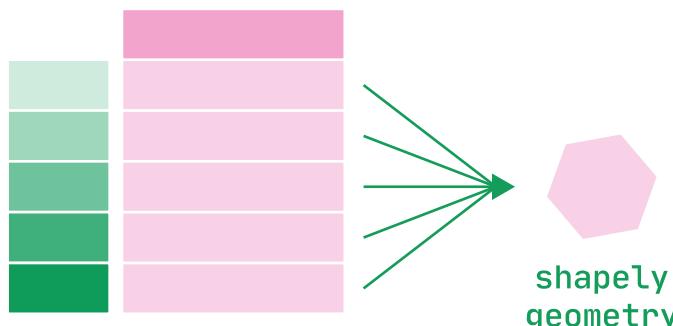
## Examples

```
>>> from shapely.geometry import Polygon, LineString, MultiPoint, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... MultiPoint([(0, 0), (0, 1)]),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (-2, 0), (0, -2)]),
... LineString([(0, 1), (1, 1)]),
... LineString([(1, 1), (3, 0)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 MULTIPOINT ((0 0), (0 1))
dtype: geometry
```

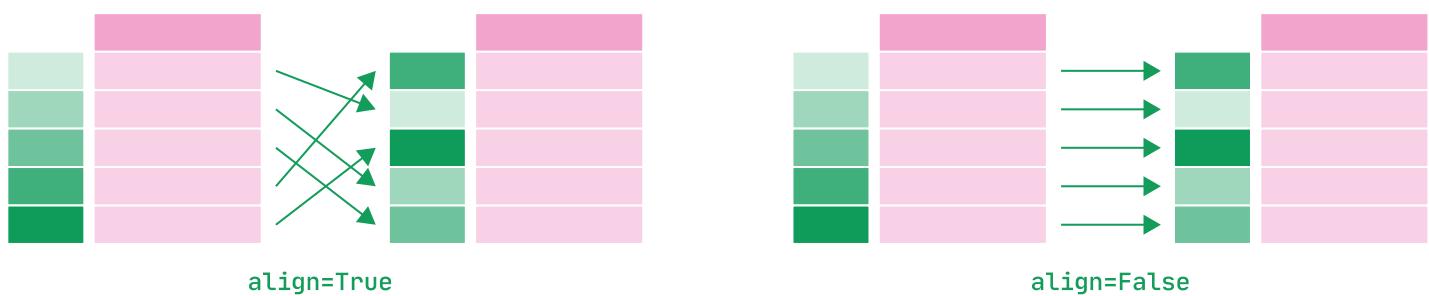
```
>>> s2
1 POLYGON ((0 0, -2 0, 0 -2, 0 0))
2 LINESTRING (0 1, 1 1)
3 LINESTRING (1 1, 3 0)
4 POINT (0 1)
dtype: geometry
```

We can check if each geometry of GeoSeries touches a single geometry:



```
>>> line = LineString([(0, 0), (-1, -2)])
>>> s.touches(line)
0 True
1 True
2 True
3 True
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.touches(s2, align=True)
0 False
1 True
2 True
3 False
4 False
dtype: bool
```

```
>>> s.touches(s2, align=False)
0 True
1 False
2 True
3 False
dtype: bool
```

# geopandas.GeoSeries.within

`GeoSeries.within(other, align=None)`

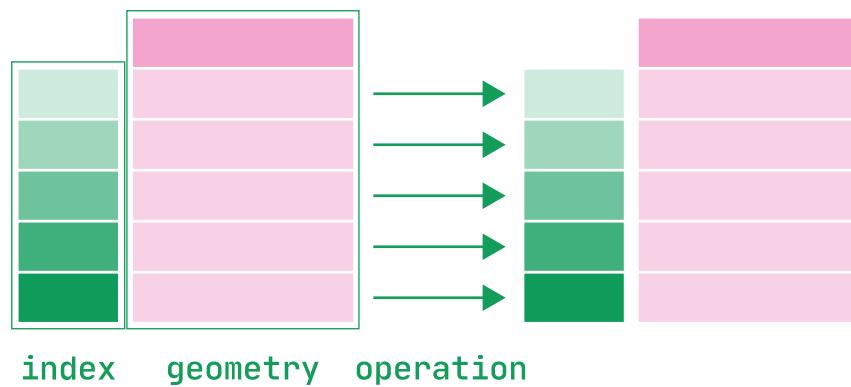
[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is within `other`.

An object is said to be within `other` if at least one of its points is located in the *interior* and no points are located in the *exterior* of the other. If either object is empty, this operation returns `False`.

This is the inverse of [`contains\(\)`](#) in the sense that the expression `a.within(b) == b.contains(a)` always evaluates to `True`.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : GeoSeries or geometric object`

The GeoSeries (elementwise) or geometric object to test if each geometry is within.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.contains`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is `within` any element of the other one.

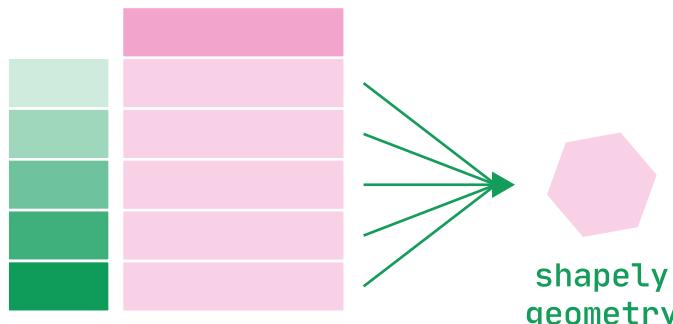
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (1, 2), (0, 2)]),
... LineString([(0, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (0, 2)]),
... LineString([(0, 0), (0, 1)]),
... Point(0, 1),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 1 2, 0 2, 0 0))
2 LINESTRING (0 0, 0 2)
3 POINT (0 1)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0 0, 1 1, 0 1, 0 0))
2 LINESTRING (0 0, 0 2)
3 LINESTRING (0 0, 0 1)
4 POINT (0 1)
dtype: geometry
```

We can check if each geometry of GeoSeries is within a single geometry:



```
>>> polygon = Polygon([(0, 0), (2, 2), (0, 2)])
>>> s.within(polygon)
0 True
1 True
2 False
3 False
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s2.within(s)
0 False
1 False
2 True
3 False
4 False
dtype: bool
```

```
>>> s2.within(s, align=False)
1 True
2 False
3 True
4 True
dtype: bool
```

# geopandas.GeoSeries.covers

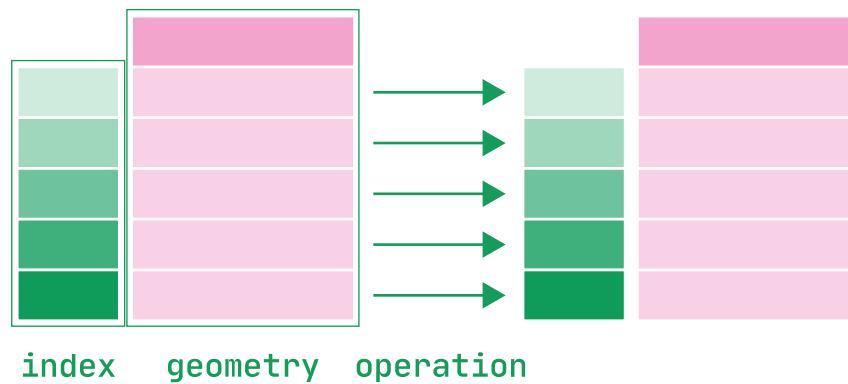
`GeoSeries.covers(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is entirely covering `other`.

An object A is said to cover another object B if no points of B lie in the exterior of A. If either object is empty, this operation returns `False`.

The operation works on a 1-to-1 row-wise manner:



See <https://lin-eat-h-inking.blogspot.com/2007/06/subtleties-of-ogc-covers-spatial.html> for reference.

## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to check is being covered.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.covered\_by`](#)

[`GeoSeries.overlaps`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries `covers` any element of the other one.

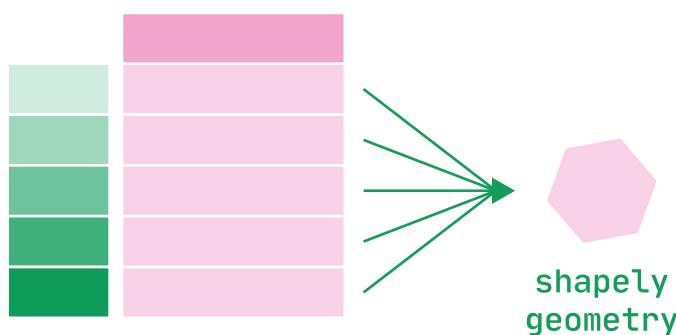
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... Point(0, 0),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0.5, 0.5), (1.5, 0.5), (1.5, 1.5), (0.5, 1.5)]),
... Polygon([(0, 0), (2, 0), (2, 2), (0, 2)]),
... LineString([(1, 1), (1.5, 1.5)]),
... Point(0, 0),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 POINT (0 0)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0.5 0.5, 1.5 0.5, 1.5 1.5, 0.5 1.5, ...
2 POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))
3 LINESTRING (1 1, 1.5 1.5)
4 POINT (0 0)
dtype: geometry
```

We can check if each geometry of GeoSeries covers a single geometry:



```
>>> poly = Polygon([(0, 0), (2, 0), (2, 2), (0, 2)])
>>> s.covers(poly)
0 True
1 False
2 False
3 False
dtype: bool
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.covers(s2, align=True)
0 False
1 False
2 False
3 False
4 False
dtype: bool
```

```
>>> s.covers(s2, align=False)
0 True
```

# geopandas.GeoSeries.covered\_by

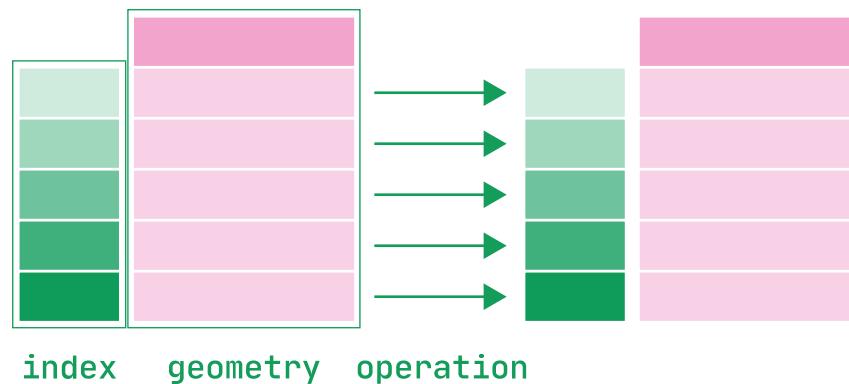
`GeoSeries.covered_by(other, align=None)`

[\[source\]](#)

Returns a `Series` of `dtype('bool')` with value `True` for each aligned geometry that is entirely covered by `other`.

An object A is said to cover another object B if no points of B lie in the exterior of A.

The operation works on a 1-to-1 row-wise manner:



See <https://lin-eat-th-inking.blogspot.com/2007/06/subtleties-of-ogc-covers-spatial.html> for reference.

## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to check is being covered.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series (bool)`

## See also

[`GeoSeries.covers`](#)

[`GeoSeries.overlaps`](#)

## Notes

This method works in a row-wise manner. It does not check if an element of one GeoSeries is `covered_by` any element of the other one.

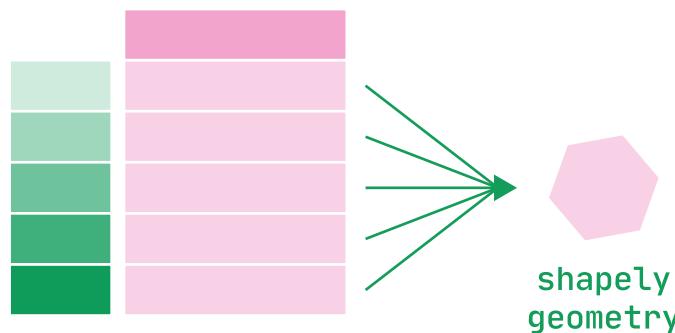
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0.5, 0.5), (1.5, 0.5), (1.5, 1.5), (0.5, 1.5)]),
... Polygon([(0, 0), (2, 0), (2, 2), (0, 2)]),
... LineString([(1, 1), (1.5, 1.5)]),
... Point(0, 0),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... Point(0, 0),
...],
... index=range(1, 5),
...)
```

```
>>> s
0 POLYGON ((0.5 0.5, 1.5 0.5, 1.5 1.5, 0.5 1.5, ...
1 POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))
2 LINESTRING (1 1, 1.5 1.5)
3 POINT (0 0)
dtype: geometry
>>>
```

```
>>> s2
1 POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))
2 POLYGON ((0 0, 2 2, 0 2, 0 0))
3 LINESTRING (0 0, 2 2)
4 POINT (0 0)
dtype: geometry
```

We can check if each geometry of GeoSeries is covered by a single geometry:



```
>>> poly = Polygon([(0, 0), (2, 0), (2, 2), (0, 2)])
>>> s.covered_by(poly)
0 True
1 True
2 True
3 True
dtype: bool
>>>
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.covered_by(s2, align=True)
0 False
1 True
2 True
3 True
4 False
dtype: bool
```

```
>>> s.covered_by(s2, align=False)
0 True
1 False
2 True
3 True
dtype: bool
```

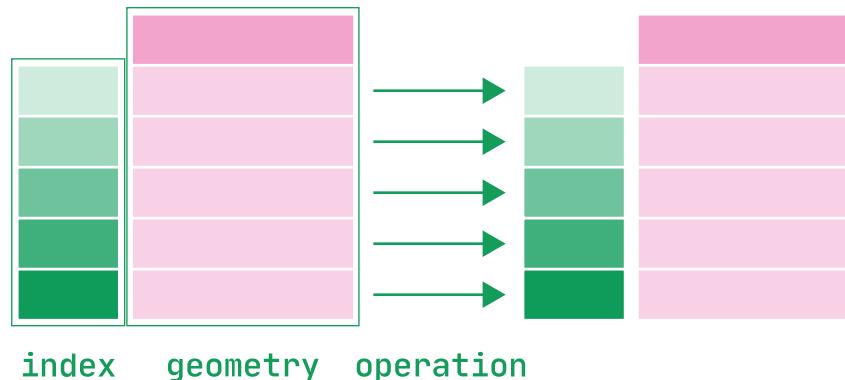
# geopandas.GeoSeries.relate

`GeoSeries.relate(other, align=None)`

[\[source\]](#)

Returns the DE-9IM intersection matrices for the geometries

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : BaseGeometry or GeoSeries`

The other geometry to computed the DE-9IM intersection matrices from.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`spatial_relations: Series of strings`

The DE-9IM intersection matrices which describe the spatial relations of the other geometry.

## Examples

```

>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 6),
...)

```

```

>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 LINESTRING (2 0, 0 2)
4 POINT (0 1)
dtype: geometry

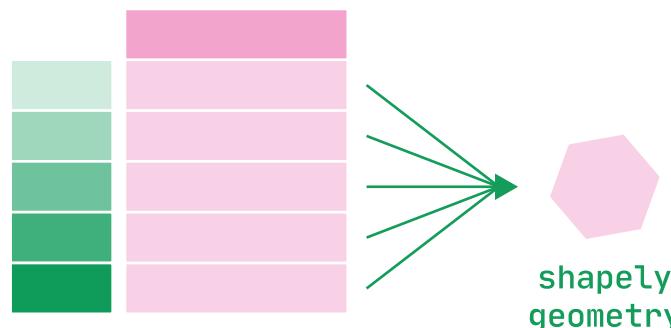
```

```

>>> s2
1 POLYGON ((0 0, 1 1, 0 1, 0 0))
2 LINESTRING (1 0, 1 3)
3 LINESTRING (2 0, 0 2)
4 POINT (1 1)
5 POINT (0 1)
dtype: geometry

```

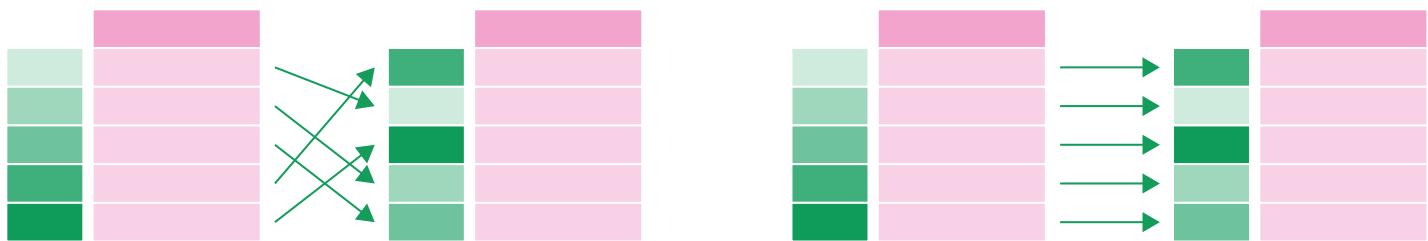
We can relate each geometry and a single shapely geometry:



```
>>> s.relate(Polygon([(0, 0), (1, 1), (0, 1)]))
0 212F11FF2
1 212F11FF2
2 F11F00212
3 F01FF0212
4 F0FFFF212
dtype: object
```

>>>

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.relate\_pattern

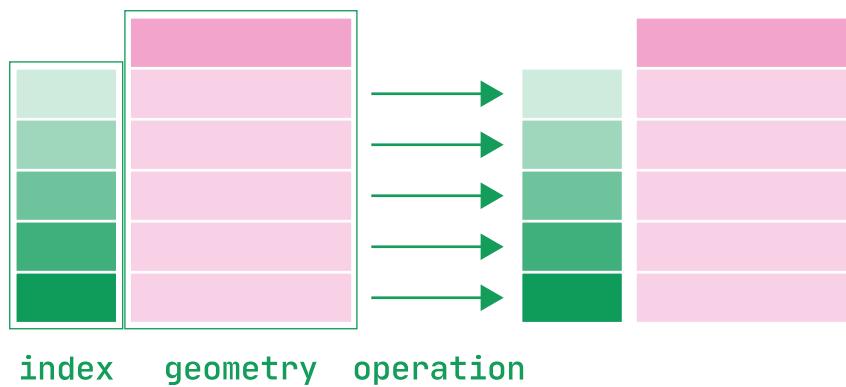
`GeoSeries.relate_pattern(other, pattern, align=None)`

[\[source\]](#)

Returns True if the DE-9IM string code for the relationship between the geometries satisfies the pattern, else False.

This function compares the DE-9IM code string for two geometries against a specified pattern. If the string matches the pattern then `True` is returned, otherwise `False`. The pattern specified can be an exact match (`0`, `1` or `2`), a boolean match (uppercase `T` or `F`), or a wildcard (`*`). For example, the pattern for the `within` predicate is `'T*F**F***'`

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : BaseGeometry or GeoSeries`

The other geometry to be tested agains the pattern.

`pattern : str`

The DE-9IM pattern to test against.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`Series`

## Examples

```

>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 6),
...)

```

```

>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 LINESTRING (2 0, 0 2)
4 POINT (0 1)
dtype: geometry

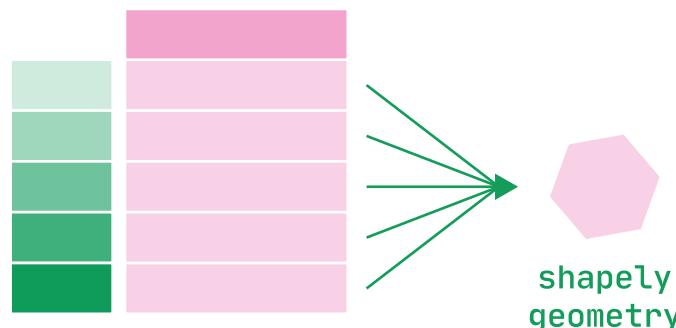
```

```

>>> s2
1 POLYGON ((0 0, 1 1, 0 1, 0 0))
2 LINESTRING (1 0, 1 3)
3 LINESTRING (2 0, 0 2)
4 POINT (1 1)
5 POINT (0 1)
dtype: geometry

```

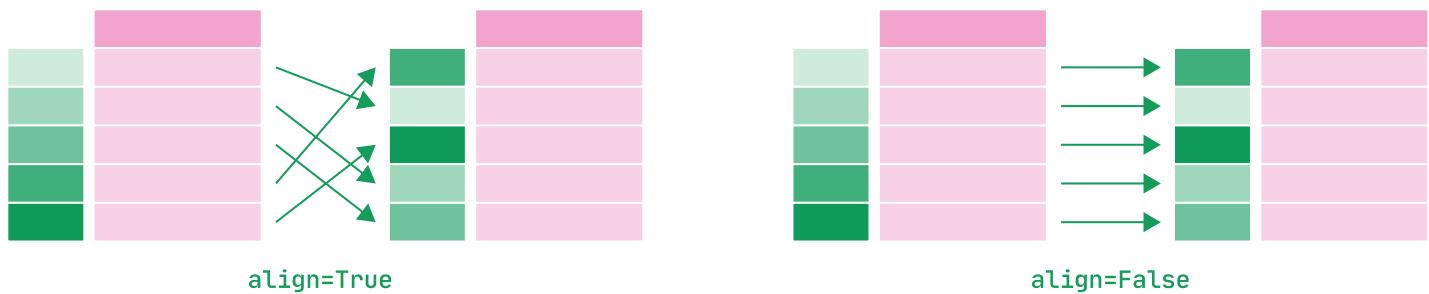
We can check the relate pattern of each geometry and a single shapely geometry:



```
>>> s.relate_pattern(Polygon([(0, 0), (1, 1), (0, 1)]), "2*T***F**")
0 True
1 True
2 False
3 False
4 False
dtype: bool
```

>>>

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.relate_pattern(s2, "TF*****T", align=True)
0 False
1 False
2 True
3 True
4 False
5 False
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.



Verifying you are human. This may take a few seconds.

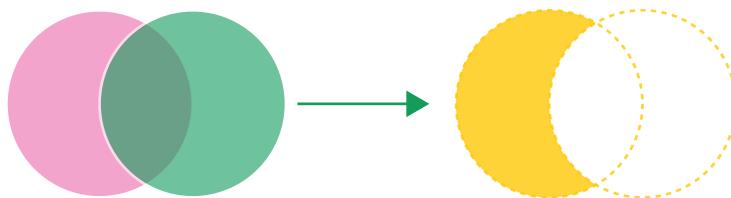
geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.difference

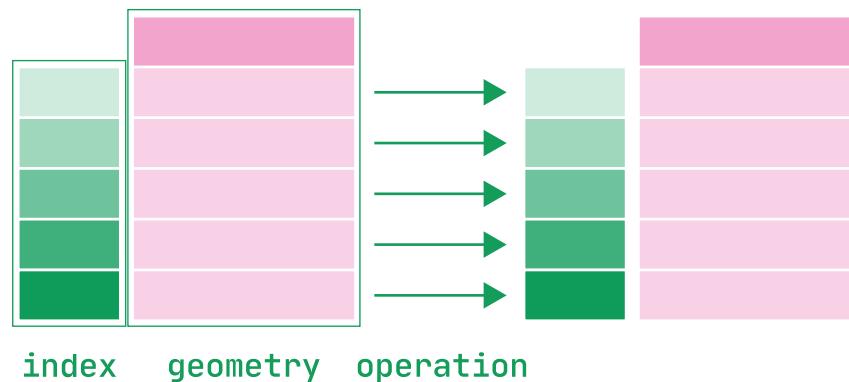
`GeoSeries.difference(other, align=None)`

[\[source\]](#)

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



The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to find the difference to.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`GeoSeries`

## See also

[`GeoSeries.symmetric\_difference`](#)

[`GeoSeries.union`](#)

[`GeoSeries.intersection`](#)

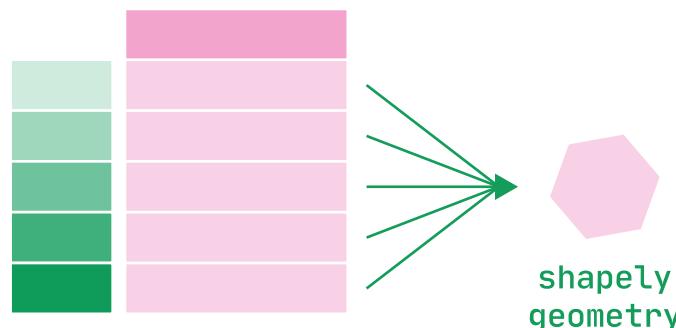
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 6),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 LINESTRING (2 0, 0 2)
4 POINT (0 1)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0 0, 1 1, 0 1, 0 0))
2 LINESTRING (1 0, 1 3)
3 LINESTRING (2 0, 0 2)
4 POINT (1 1)
5 POINT (0 1)
dtype: geometry
```

We can do difference of each geometry and a single shapely geometry:



```
>>> s.difference(Polygon([(0, 0), (1, 1), (0, 1)]))
0 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
1 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
2 LINESTRING (1 1, 2 2)
3 MULTILINESTRING ((2 0, 1 1), (1 1, 0 2))
4 POINT EMPTY
dtype: geometry
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.difference(s2, align=True)
0 None
1 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
2 MULTILINESTRING ((0 0, 1 1), (1 1, 2 2))
3 LINESTRING EMPTY
4 POINT (0 1)
5 None
dtype: geometry
```

```
>>> s.difference(s2, align=False)
```

Built with the [PyData Sphinx Theme 0.15.4](#).

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

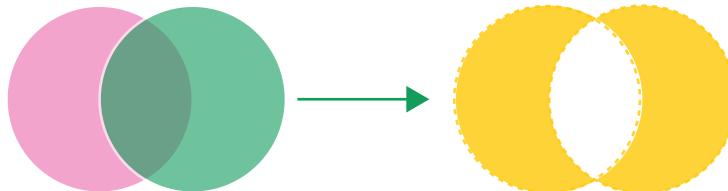
# geopandas.GeoSeries.symmetric\_difference

`GeoSeries.symmetric_difference(other, align=None)`

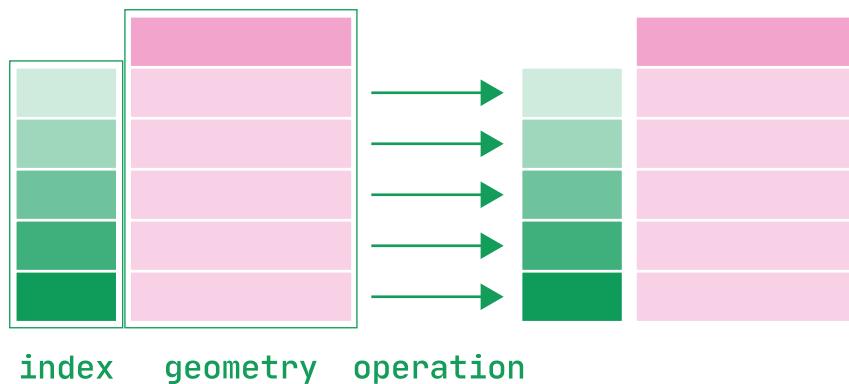
[\[source\]](#)

Returns a `GeoSeries` of the symmetric difference of points in each aligned geometry with `other`.

For each geometry, the symmetric difference consists of points in the geometry not in `other`, and points in `other` not in the geometry.



The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to find the symmetric difference to.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`GeoSeries`

## See also

[`GeoSeries.difference`](#)

[`GeoSeries.union`](#)

[`GeoSeries.intersection`](#)

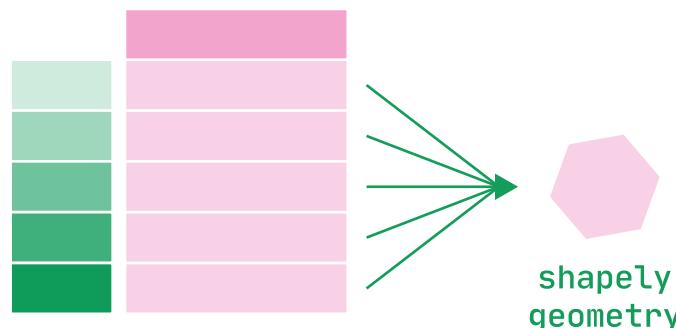
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 6),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 LINESTRING (2 0, 0 2)
4 POINT (0 1)
dtype: geometry
```

```
>>> s2
1 POLYGON ((0 0, 1 1, 0 1, 0 0))
2 LINESTRING (1 0, 1 3)
3 LINESTRING (2 0, 0 2)
4 POINT (1 1)
5 POINT (0 1)
dtype: geometry
```

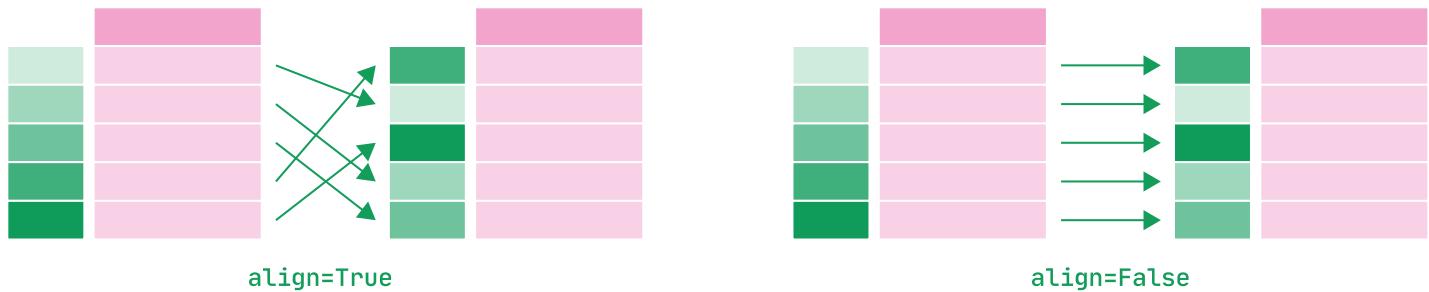
We can do symmetric difference of each geometry and a single shapely geometry:



```
>>> s.symmetric_difference(Polygon([(0, 0), (1, 1), (0, 1)]))
0 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
1 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
2 GEOMETRYCOLLECTION (POLYGON ((0 0, 0 1, 1 1, 0...
3 GEOMETRYCOLLECTION (POLYGON ((0 0, 0 1, 1 1, 0...
4 POLYGON ((0 1, 1 1, 0 0, 0 1))
dtype: geometry
```

>>>

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.symmetric_difference(s2, align=True)
0 None
1 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
2 MULTILINESTRING ((0 0, 1 1), (1 1, 2 2), (1 0, ...
3 LINESTRING EMPTY
4 MULTIPOINT ((0 1), (1 1))
5 None
dtype: geometry
```

>>>

```
>>> s.symmetric_difference(s2, align=False)
0 POLYGON ((0 2, 2 2, 1 1, 0 1, 0 2))
1 GEOMETRYCOLLECTION (POLYGON ((0 0, 0 2, 1 2, 2...
2 MULTILINESTRING ((0 0, 1 1), (1 1, 2 2), (2 0, ...
3 LINESTRING (2 0, 0 2))
```

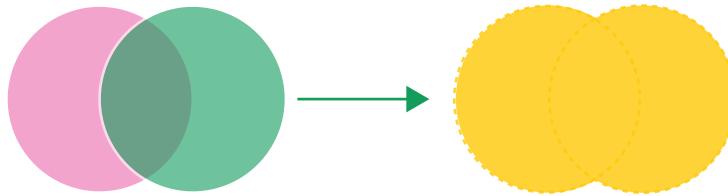
>>>

# geopandas.GeoSeries.union

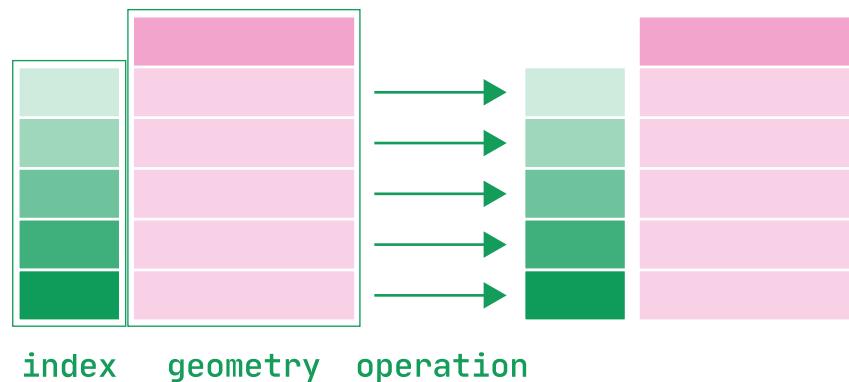
`GeoSeries.union(other, align=None)`

[\[source\]](#)

Returns a `GeoSeries` of the union of points in each aligned geometry with `other`.



The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to find the union with.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`GeoSeries`

## See also

[`GeoSeries.symmetric\_difference`](#)

[`GeoSeries.difference`](#)

[`GeoSeries.intersection`](#)

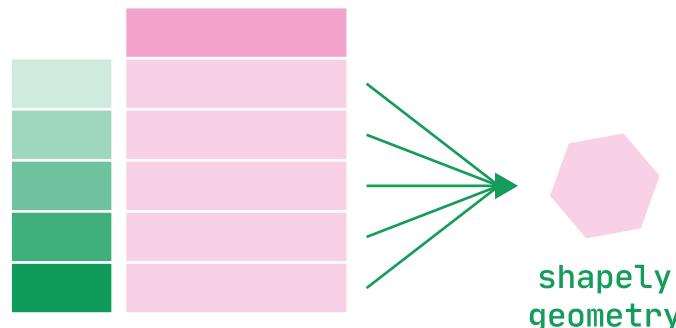
## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(1, 0), (1, 3)]),
... LineString([(2, 0), (0, 2)]),
... Point(1, 1),
... Point(0, 1),
...],
... index=range(1, 6),
...)
```

```
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 LINESTRING (2 0, 0 2)
4 POINT (0 1)
dtype: geometry
>>>
```

```
>>> s2
1 POLYGON ((0 0, 1 1, 0 1, 0 0))
2 LINESTRING (1 0, 1 3)
3 LINESTRING (2 0, 0 2)
4 POINT (1 1)
5 POINT (0 1)
dtype: geometry
```

We can do union of each geometry and a single shapely geometry:



&gt;&gt;&gt;

```
>>> s.union(Polygon([(0, 0), (1, 1), (0, 1)]))
0 POLYGON ((0 0, 0 1, 0 2, 2 2, 1 1, 0 0))
1 POLYGON ((0 0, 0 1, 0 2, 2 2, 1 1, 0 0))
2 GEOMETRYCOLLECTION (POLYGON ((0 0, 0 1, 1 1, 0...
3 GEOMETRYCOLLECTION (POLYGON ((0 0, 0 1, 1 1, 0...
4 POLYGON ((0 1, 1 1, 0 0, 0 1)))
dtype: geometry
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s.union(s2, align=True)
0 None
1 POLYGON ((0 0, 0 1, 0 2, 2 2, 1 1, 0 0))
2 MULTILINESTRING ((0 0, 1 1), (1 1, 2 2), (1 0, ...
3 LINESTRING (2 0, 0 2)
4 MULTIPOINT ((0 1), (1 1))
5 None
dtype: geometry
```

```
>>> s.union(s2, align=False)
0 POLYGON ((0 0, 0 1, 0 2, 2 2, 1 1, 0 0))
```

# geopandas.GeoSeries.buffer

`GeoSeries.buffer(distance, resolution=16, cap_style='round',`

`join_style='round', mitre_limit=5.0, single_sided=False, **kwargs)`

[\[source\]](#)

Returns a `GeoSeries` of geometries representing all points within a given `distance` of each geometric object.

Computes the buffer of a geometry for positive and negative buffer distance.

The buffer of a geometry is defined as the Minkowski sum (or difference, for negative distance) of the geometry with a circle with radius equal to the absolute value of the buffer distance.

The buffer operation always returns a polygonal result. The negative or zero-distance buffer of lines and points is always empty.

## Parameters:

**distance : float, np.array, pd.Series**

The radius of the buffer in the Minkowski sum (or difference). If np.array or pd.Series are used then it must have same length as the GeoSeries.

**resolution : int (optional, default 16)**

The resolution of the buffer around each vertex. Specifies the number of linear segments in a quarter circle in the approximation of circular arcs.

**cap\_style : {'round', 'square', 'flat'}, default 'round'**

Specifies the shape of buffered line endings. `'round'` results in circular line endings (see `resolution`). Both `'square'` and `'flat'` result in rectangular line endings, `'flat'` will end at the original vertex, while `'square'` involves adding the buffer width.

**join\_style : {'round', 'mitre', 'bevel'}, default 'round'**

Specifies the shape of buffered line midpoints. `'round'` results in rounded shapes. `'bevel'` results in a beveled edge that touches the original vertex. `'mitre'` results in a single vertex that is beveled depending on the `mitre_limit` parameter.

**mitre\_limit : float, default 5.0**

Crops off `'mitre'`-style joins if the point is displaced from the buffered vertex by more than this limit.

**single\_sided : bool, default False**

Only buffer at one side of the geometry.

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(0, 0),
... LineString([(1, -1), (1, 0), (2, 0), (2, 1)]),
... Polygon([(3, -1), (4, 0), (3, 1)])
...]
...)
>>> s
0 POINT (0 0)
1 LINESTRING (1 -1, 1 0, 2 0, 2 1)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.buffer(0.2)
0 POLYGON ((0.2 0, 0.19904 -0.0196, 0.19616 -0.0...
1 POLYGON ((0.8 0, 0.80096 0.0196, 0.80384 0.039...
2 POLYGON ((2.8 -1, 2.8 1, 2.80096 1.0196, 2.803...
dtype: geometry
```

Further specification as `join_style` and `cap_style` are shown in the following illustration:

( [Source code](#),  [png](#),  [hires.png](#),  [pdf](#))



© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.centroid

*property* `GeoSeries.centroid`

[\[source\]](#)

Returns a `GeoSeries` of points representing the centroid of each geometry.

Note that centroid does not have to be on or within original geometry.

## See also

[`GeoSeries.representative\_point`](#)

point guaranteed to be within each geometry

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.centroid
0 POINT (0.33333 0.66667)
1 POINT (0.70711 0.5)
2 POINT (0 0)
dtype: geometry
```

# geopandas.GeoSeries.concave\_hull

`GeoSeries.concave_hull(ratio=0.0, allow_holes=False)`

[\[source\]](#)

Returns a `GeoSeries` of geometries representing the concave hull of each geometry.

The concave hull of a geometry is the smallest concave *Polygon* containing all the points in each geometry, unless the number of points in the geometric object is less than three. For two points, the concave hull collapses to a *LineString*; for 1, a *Point*.

The hull is constructed by removing border triangles of the Delaunay Triangulation of the points as long as their “size” is larger than the maximum edge length ratio and optionally allowing holes. The edge length factor is a fraction of the length difference between the longest and shortest edges in the Delaunay Triangulation of the input points. For further information on the algorithm used, see [https://libgeos.org/doxygen/classgeos\\_1\\_1algorithm\\_1\\_1hull\\_1\\_1ConcaveHull.html](https://libgeos.org/doxygen/classgeos_1_1algorithm_1_1hull_1_1ConcaveHull.html)

## Parameters:

`ratio : float, (optional, default 0.0)`

Number in the range [0, 1]. Higher numbers will include fewer vertices in the hull.

`allow_holes : bool, (optional, default False)`

If set to True, the concave hull may have holes.

## See also

[GeoSeries.convex\\_hull](#)

convex hull geometry

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point, MultiPoint
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... MultiPoint([(0, 0), (1, 1), (0, 1), (1, 0), (0.5, 0.5)]),
... MultiPoint([(0, 0), (1, 1)]),
... Point(0, 0),
...],
... crs=3857
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 MULTIPOLY ((0 0), (1 1), (0 1), (1 0), (0.5 0...))
3 MULTIPOLY ((0 0), (1 1))
4 POINT (0 0)
dtype: geometry
```

```
>>> s.concave_hull()
0 POLYGON ((0 1, 1 1, 0 0, 0 1))
1 POLYGON ((0 0, 1 1, 1 0, 0 0))
2 POLYGON ((0.5 0.5, 0 1, 1 1, 1 0, 0 0, 0.5 0.5))
3 LINESTRING (0 0, 1 1)
4 POINT (0 0)
dtype: geometry
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.convex\_hull

**property** `GeoSeries.convex_hull`

[\[source\]](#)

Returns a `GeoSeries` of geometries representing the convex hull of each geometry.

The convex hull of a geometry is the smallest convex *Polygon* containing all the points in each geometry, unless the number of points in the geometric object is less than three. For two points, the convex hull collapses to a *LineString*; for 1, a *Point*.

## See also

[`GeoSeries.concave\_hull`](#)

concave hull geometry

[`GeoSeries.envelope`](#)

bounding rectangle geometry

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point, MultiPoint
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... MultiPoint([(0, 0), (1, 1), (0, 1), (1, 0), (0.5, 0.5)]),
... MultiPoint([(0, 0), (1, 1)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 MULTIPOINT ((0 0), (1 1), (0 1), (1 0), (0.5 0...))
3 MULTIPOINT ((0 0), (1 1))
4 POINT (0 0)
dtype: geometry
```

```
>>> s.convex_hull
0 POLYGON ((0 0, 0 1, 1 1, 0 0))
1 POLYGON ((0 0, 1 1, 1 0, 0 0))
2 POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))
3 LINESTRING (0 0, 1 1)
4 POINT (0 0)
dtype: geometry
```





Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.extract\_unique\_point

## GeoSeries.extract\_unique\_points()

[\[source\]](#)

Returns a [GeoSeries](#) of MultiPoints representing all distinct vertices of an input geometry.

### See also

[GeoSeries.get\\_coordinates](#)

extract coordinates as a [DataFrame](#)

## Examples

```
>>> from shapely import LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (0, 0), (1, 1), (1, 1)]),
... Polygon([(0, 0), (0, 0), (1, 1), (1, 1)])
...],
...)
>>> s
0 LINESTRING (0 0, 0 0, 1 1, 1 1)
1 POLYGON ((0 0, 0 0, 1 1, 1 1, 0 0))
dtype: geometry
```

```
>>> s.extract_unique_points()
0 MULTIPOINT ((0 0), (1 1))
1 MULTIPOINT ((0 0), (1 1))
dtype: geometry
```

# geopandas.GeoSeries.force\_2d

## GeoSeries.force\_2d()

[\[source\]](#)

Forces the dimensionality of a geometry to 2D.

Removes the additional Z coordinate dimension from all geometries.

### Returns:

GeoSeries

## Examples

```
>>> from shapely import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Point(0.5, 2.5, 0),
... LineString([(1, 1, 1), (0, 1, 3), (1, 0, 2)]),
... Polygon([(0, 0, 0), (0, 10, 0), (10, 10, 0)]),
...],
...)
>>> s
0 POINT Z (0.5 2.5 0)
1 LINESTRING Z (1 1 1, 0 1 3, 1 0 2)
2 POLYGON Z ((0 0 0, 0 10 0, 10 10 0, 0 0 0))
dtype: geometry
```

```
>>> s.force_2d()
0 POINT (0.5 2.5)
1 LINESTRING (1 1, 0 1, 1 0)
2 POLYGON ((0 0, 0 10, 10 10, 0 0))
dtype: geometry
```

# geopandas.GeoSeries.force\_3d

`GeoSeries.force_3d(z=0)`

[\[source\]](#)

Forces the dimensionality of a geometry to 3D.

2D geometries will get the provided Z coordinate; 3D geometries are unchanged (unless their Z coordinate is `np.nan`).

Note that for empty geometries, 3D is only supported since GEOS 3.9 and then still only for simple geometries (non-collections).

## Parameters:

`z : float | array_like (default 0)`

Z coordinate to be assigned

## Returns:

`GeoSeries`

## Examples

```
>>> from shapely import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Point(1, 2),
... Point(0.5, 2.5, 2),
... LineString([(1, 1), (0, 1), (1, 0)]),
... Polygon([(0, 0), (0, 10), (10, 10)]),
...],
...)
>>> s
0 POINT (1 2)
1 POINT Z (0.5 2.5 2)
2 LINESTRING (1 1, 0 1, 1 0)
3 POLYGON ((0 0, 0 10, 10 10, 0 0))
dtype: geometry
```

```
>>> s.force_3d()
0 POINT Z (1 2 0)
1 POINT Z (0.5 2.5 2)
2 LINESTRING Z (1 1 0, 0 1 0, 1 0 0)
3 POLYGON Z ((0 0 0, 0 10 0, 10 10 0, 0 0 0))
dtype: geometry
```

Z coordinate can be specified as scalar:

```
>>> s.force_3d(4)
0 POINT Z (1 2 4)
1 POINT Z (0.5 2.5 2)
2 LINESTRING Z (1 1 4, 0 1 4, 1 0 4)
3 POLYGON Z ((0 0 4, 0 10 4, 10 10 4, 0 0 4))
dtype: geometry
```

Or as an array-like (one value per geometry):

```
>>> s.force_3d(range(4))
0 POINT Z (1 2 0)
1 POINT Z (0.5 2.5 2)
2 LINESTRING Z (1 1 2, 0 1 2, 1 0 2)
3 POLYGON Z ((0 0 3, 0 10 3, 10 10 3, 0 0 3))
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.make\_valid

## GeoSeries.make\_valid()

[\[source\]](#)

Repairs invalid geometries.

Returns a `GeoSeries` with valid geometries. If the input geometry is already valid, then it will be preserved. In many cases, in order to create a valid geometry, the input geometry must be split into multiple parts or multiple geometries. If the geometry must be split into multiple parts of the same type to be made valid, then a multi-part geometry will be returned (e.g. a MultiPolygon). If the geometry must be split into multiple parts of different types to be made valid, then a GeometryCollection will be returned.

## Examples

```
>>> from shapely.geometry import MultiPolygon, Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (0, 2), (1, 1), (2, 2), (2, 0), (1, 1), (0, 0)]),
... Polygon([(0, 2), (0, 1), (2, 0), (0, 0), (0, 2)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
...],
...)
>>> s
0 POLYGON ((0 0, 0 2, 1 1, 2 2, 2 0, 1 1, 0 0))
1 POLYGON ((0 2, 0 1, 2 0, 0 0, 0 2))
2 LINESTRING (0 0, 1 1, 1 0)
dtype: geometry
```

```
>>> s.make_valid()
0 MULTIPOLYGON (((1 1, 0 0, 0 2, 1 1)), ((2 0, 1...
1 GEOMETRYCOLLECTION (POLYGON ((2 0, 0 0, 0 1, 2...
2 LINESTRING (0 0, 1 1, 1 0))
dtype: geometry
```

# geopandas.GeoSeries.minimum\_bounding\_

## GeoSeries.minimum\_bounding\_circle()

[\[source\]](#)

Returns a `GeoSeries` of geometries representing the minimum bounding circle that encloses each geometry.

### See also

[GeoSeries.convex\\_hull](#)

convex hull geometry

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.minimum_bounding_circle()
0 POLYGON ((1.20711 0.5, 1.19352 0.36205, 1.1532...
1 POLYGON ((1.20711 0.5, 1.19352 0.36205, 1.1532...
2 POINT (0 0)
dtype: geometry
```

# geopandas.GeoSeries.minimum\_clearance

## GeoSeries.minimum\_clearance()

[\[source\]](#)

Returns a `Series` containing the minimum clearance distance, which is the smallest distance by which a vertex of the geometry could be moved to produce an invalid geometry.

If no minimum clearance exists for a geometry (for example, a single point, or an empty geometry), infinity is returned.

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(0, 0), (1, 1), (3, 2)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 3 2)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.minimum_clearance()
0 0.707107
1 1.414214
2 inf
dtype: float64
```

# geopandas.GeoSeries.minimum\_rotated\_rectangle()

[\[source\]](#)

Returns a `GeoSeries` of the general minimum bounding rectangle that contains the object.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

## See also

[GeoSeries.envelope](#)

bounding rectangle

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point, MultiPoint
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... MultiPoint([(0, 0), (1, 1)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 MULTIPOLY ((0 0), (1 1))
3 POINT (0 0)
dtype: geometry
```

```
>>> s.minimum_rotated_rectangle()
0 POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))
1 POLYGON ((1 1, 1 0, 0 0, 0 1, 1 1))
2 LINESTRING (0 0, 1 1)
3 POINT (0 0)
dtype: geometry
```

# geopandas.GeoSeries.normalize

## GeoSeries.normalize()

[\[source\]](#)

Returns a `GeoSeries` of normalized geometries to normal form (or canonical form).

This method orders the coordinates, rings of a polygon and parts of multi geometries consistently.

Typically useful for testing purposes (for example in combination with `equals_exact`).

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0, 0),
...],
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.normalize()
0 POLYGON ((0 0, 0 1, 1 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

# geopandas.GeoSeries.remove\_repeated\_points

`GeoSeries.remove_repeated_points(tolerance=0.0)`

[\[source\]](#)

Returns a `GeoSeries` containing a copy of the input geometry with repeated points removed.

From the start of the coordinate sequence, each next point within the tolerance is removed.

Removing repeated points with a non-zero tolerance may result in an invalid geometry being returned.

## Parameters:

`tolerance : float, default 0.0`

Remove all points within this distance of each other. Use 0.0 to remove only exactly repeated points (the default).

## Examples

```
>>> from shapely import LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (0, 0), (1, 0)]),
... Polygon([(0, 0), (0, 0.5), (0, 1), (0.5, 1), (0, 0)]),
...],
...)
>>> s
0 LINESTRING (0 0, 0 0, 1 0)
1 POLYGON ((0 0, 0 0.5, 0 1, 0.5 1, 0 0))
dtype: geometry
```

```
>>> s.remove_repeated_points(tolerance=0.0)
0 LINESTRING (0 0, 1 0)
1 POLYGON ((0 0, 0 0.5, 0 1, 0.5 1, 0 0))
dtype: geometry
```

# geopandas.GeoSeries.reverse

## GeoSeries.reverse()

[\[source\]](#)

Returns a [GeoSeries](#) with the order of coordinates reversed.

### See also

[GeoSeries.normalize](#)

normalize order of coordinates

## Examples

```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... LineString([(0, 0), (1, 1), (1, 0)]),
... Point(0, 0),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 LINESTRING (0 0, 1 1, 1 0)
2 POINT (0 0)
dtype: geometry
```

```
>>> s.reverse()
0 POLYGON ((0 0, 0 1, 1 1, 0 0))
1 LINESTRING (1 0, 1 1, 0 0)
2 POINT (0 0)
dtype: geometry
```

# geopandas.GeoSeries.sample\_points

`GeoSeries.sample_points(size, method='uniform', seed=None, rng=None, **kwargs)`

[\[source\]](#)

Sample points from each geometry.

Generate a MultiPoint per each geometry containing points sampled from the geometry. You can either sample randomly from a uniform distribution or use an advanced sampling algorithm from the `pointpats` package.

For polygons, this samples within the area of the polygon. For lines, this samples along the length of the linestring. For multi-part geometries, the weights of each part are selected according to their relevant attribute (area for Polygons, length for LineStrings), and then points are sampled from each part.

Any other geometry type (e.g. Point, GeometryCollection) is ignored, and an empty MultiPoint geometry is returned.

## Parameters:

**size : int | array-like**

The size of the sample requested. Indicates the number of samples to draw from each geometry. If an array of the same length as a GeoSeries is passed, it denotes the size of a sample per geometry.

**method : str, default “uniform”**

The sampling method. `uniform` samples uniformly at random from a geometry using `numpy.random.uniform`. Other allowed strings (e.g. `"cluster_poisson"`) denote sampling function name from the `pointpats.random` module (see <http://pysal.org/pointpats/api.html#random-distributions>). Pointpats methods are implemented for (Multi)Polygons only and will return an empty MultiPoint for other geometry types.

**rng : {None, int, array\_like[ints], SeedSequence, BitGenerator, Generator}, optional**

A random generator or seed to initialize the numpy BitGenerator. If None, then fresh, unpredictable entropy will be pulled from the OS.

**\*\*kwargs : dict**

Options for the pointpats sampling algorithms.

## Returns:

**GeoSeries**

Points sampled within (or along) each geometry.

## Examples

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [
... Polygon([(1, -1), (1, 0), (0, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
```

```
>>> s.sample_points(size=10)
0 MULTIPOLY ((0.1045 -0.10294), (0.35249 -0.264...
1 MULTIPOLY ((3.03261 -0.43069), (3.10068 0.114...
Name: sampled_points, dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.segmentize

`GeoSeries.segmentize(max_segment_length)`

[\[source\]](#)

Returns a `GeoSeries` with vertices added to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment. Only linear components of input geometries are densified; other geometries are returned unmodified.

## Parameters:

`max_segment_length : float | array-like`

Additional vertices will be added so that all line segments are no longer than this value. Must be greater than 0.

## Returns:

`GeoSeries`

## Examples

```
>>> from shapely.geometry import Polygon, LineString
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (0, 10)]),
... Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]),
...],
...)
>>> s
0 LINESTRING (0 0, 0 10)
1 POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))
dtype: geometry
```

```
>>> s.segmentize(max_segment_length=5)
0 LINESTRING (0 0, 0 5, 0 10)
1 POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0...
dtype: geometry
```

# geopandas.GeoSeries.shortest\_line

`GeoSeries.shortest_line(other, align=None)`

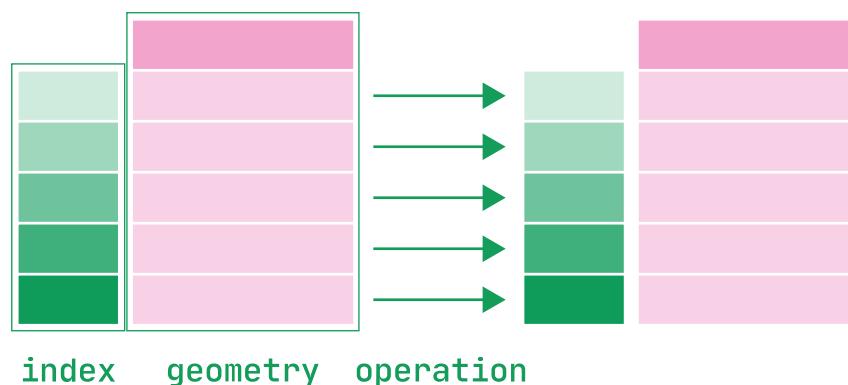
[\[source\]](#)

Returns the shortest two-point line between two geometries.

The resulting line consists of two points, representing the nearest points between the geometry pair.

The line always starts in the first geometry a and ends in he second geometry b. The endpoints of the line will not necessarily be existing vertices of the input geometries a and b, but can also be a point along a line segment.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to find the shortest line with.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

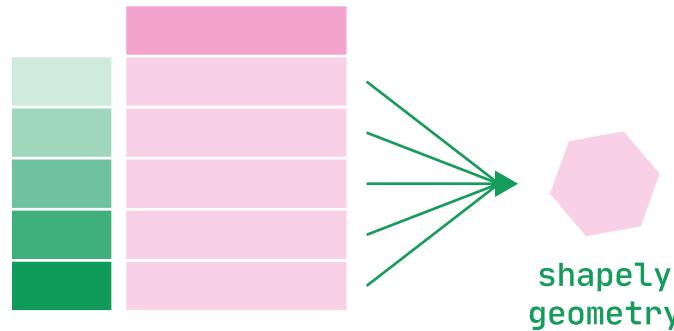
## Returns:

`GeoSeries`

## Examples

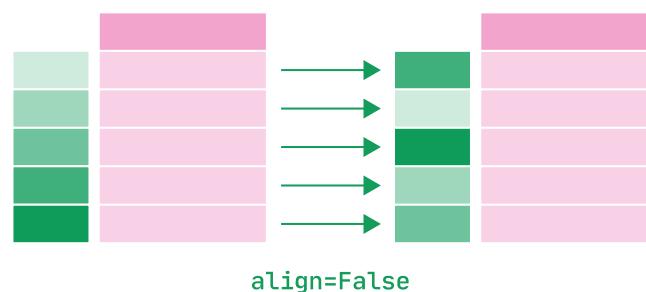
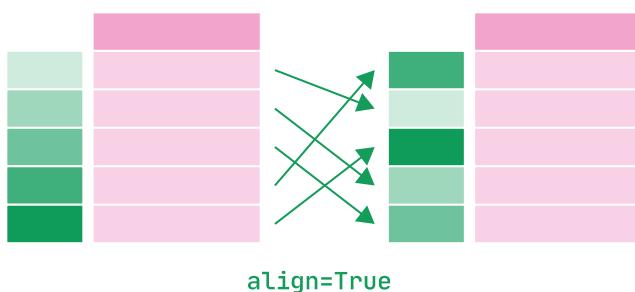
```
>>> from shapely.geometry import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (2, 2), (0, 2)]),
... Polygon([(0, 0), (2, 2), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
... Point(0, 1),
...],
...)
>>> s
0 POLYGON ((0 0, 2 2, 0 2, 0 0))
1 POLYGON ((0 0, 2 2, 0 2, 0 0))
2 LINESTRING (0 0, 2 2)
3 LINESTRING (2 0, 0 2)
4 POINT (0 1)
dtype: geometry
```

We can also do intersection of each geometry and a single shapely geometry:



```
>>> p = Point(3, 3)
>>> s.shortest_line(p)
0 LINESTRING (2 2, 3 3)
1 LINESTRING (2 2, 3 3)
2 LINESTRING (2 2, 3 3)
3 LINESTRING (1 1, 3 3)
4 LINESTRING (0 1, 3 3)
dtype: geometry
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices than the one below. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s2 = geopandas.GeoSeries(
... [
... Polygon([(0.5, 0.5), (1.5, 0.5), (1.5, 1.5), (0.5, 1.5)]),
... Point(3, 1),
... LineString([(1, 0), (2, 0)]),
... Point(10, 15),
... Point(0, 1),
...],
... index=range(1, 6),
...)
```

```
>>> s.shortest_line(s2, align=True)
0 None
1 LINESTRING (0.5 0.5, 0.5 0.5)
2 LINESTRING (2 2, 3 1)
3 LINESTRING (2 0, 2 0)
4 LINESTRING (0 1, 10 15)
5 None
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.simplify

`GeoSeries.simplify(tolerance, preserve_topology=True)`

[\[source\]](#)

Returns a `GeoSeries` containing a simplified representation of each geometry.

The algorithm (Douglas-Peucker) recursively splits the original line into smaller parts and connects these parts' endpoints by a straight line. Then, it removes all points whose distance to the straight line is smaller than *tolerance*. It does not move any points and it always preserves endpoints of the original line or polygon. See <http://shapely.readthedocs.io/en/latest/manual.html#object.simplify> for details

## Parameters:

**tolerance : float**

All parts of a simplified geometry will be no more than *tolerance* distance from the original. It has the same units as the coordinate reference system of the GeoSeries. For example, using *tolerance=100* in a projected CRS with meters as units means a distance of 100 meters in reality.

**preserve\_topology: bool (default True)**

`False` uses a quicker algorithm, but may produce self-intersecting or otherwise invalid geometries.

## Notes

Invalid geometric objects may result from simplification that does not preserve topology and simplification may be sensitive to the order of coordinates: two geometries differing only in order of coordinates may be simplified differently.

## Examples

```
>>> from shapely.geometry import Point, LineString
>>> s = geopandas.GeoSeries(
... [Point(0, 0).buffer(1), LineString([(0, 0), (1, 10), (0, 20)])]
...)
>>> s
0 POLYGON ((1 0, 0.99518 -0.09802, 0.98079 -0.19...
1 LINESTRING (0 0, 1 10, 0 20)
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.snap

`GeoSeries.snap(other, tolerance, align=None)`

[\[source\]](#)

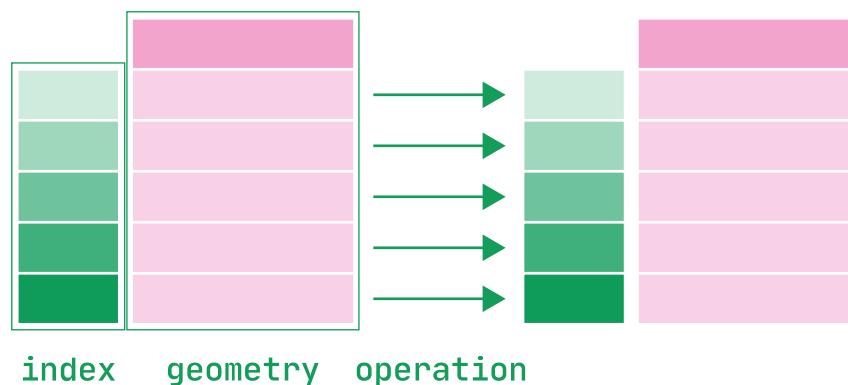
Snaps an input geometry to reference geometry's vertices.

Vertices of the first geometry are snapped to vertices of the second. geometry, returning a new geometry; the input geometries are not modified. The result geometry is the input geometry with the vertices snapped. If no snapping occurs then the input geometry is returned unchanged. The tolerance is used to control where snapping is performed.

Where possible, this operation tries to avoid creating invalid geometries; however, it does not guarantee that output geometries will be valid. It is the responsibility of the caller to check for and handle invalid geometries.

Because too much snapping can result in invalid geometries being created, heuristics are used to determine the number and location of snapped vertices that are likely safe to snap. These heuristics may omit some potential snaps that are otherwise within the tolerance.

The operation works in a 1-to-1 row-wise manner:



## Parameters:

**other : *GeoSeries or geometric object***

The Geoseries (elementwise) or geometric object to snap to.

**tolerance : *float or array like***

Maximum distance between vertices that shall be snapped

**align : *bool | None (default None)***

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

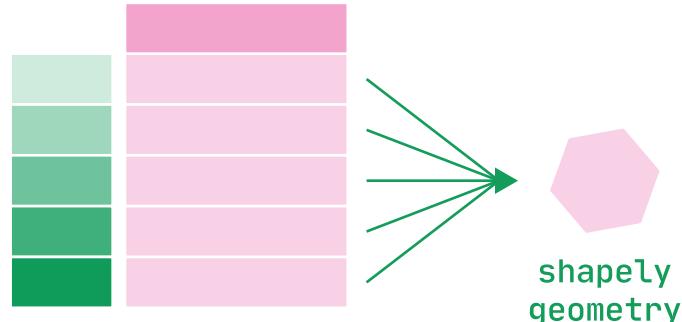
`GeoSeries`

## Examples

```
>>> from shapely import Polygon, LineString, Point
>>> s = geopandas.GeoSeries(
... [
... Point(0.5, 2.5),
... LineString([(0.1, 0.1), (0.49, 0.51), (1.01, 0.89)]),
... Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)]),
...],
...)
>>> s
0 POINT (0.5 2.5)
1 LINESTRING (0.1 0.1, 0.49 0.51, 1.01 0.89)
2 POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))
dtype: geometry
```

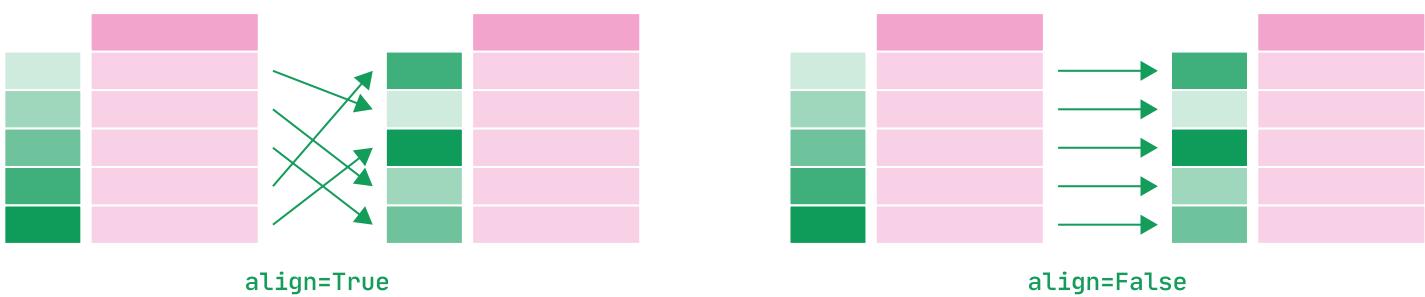
```
>>> s2 = geopandas.GeoSeries(
... [
... Point(0, 2),
... LineString([(0, 0), (0.5, 0.5), (1.0, 1.0)]),
... Point(8, 10),
...],
... index=range(1, 4),
...)
>>> s2
1 POINT (0 2)
2 LINESTRING (0 0, 0.5 0.5, 1 1)
3 POINT (8 10)
dtype: geometry
```

We can snap each geometry to a single shapely geometry:



```
>>> s.snap(Point(0, 2), tolerance=1)
0 POINT (0 2)
1 LINESTRING (0.1 0.1, 0.49 0.51, 1.01 0.89)
2 POLYGON ((0 0, 0 2, 0 10, 10 10, 10 0, 0 0))
dtype: geometry
```

We can also snap two GeoSeries to each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and snap elements with the same index using `align=True` or ignore index and snap elements based on their matching order using `align=False`:



```
>>> s.snap(s2, tolerance=1, align=True)
0 None
1 LINESTRING (0.1 0.1, 0.49 0.51, 1.01 0.89)
2 POLYGON ((0.5 0.5, 1 1, 0 10, 10 10, 10 0, 0.5...
3 None
dtype: geometry
```

```
>>> s.snap(s2, tolerance=1, align=False)
0 POINT (0 2)
1 LINESTRING (0 0, 0.5 0.5, 1 1)
2 POLYGON ((0 0, 0 10, 8 10, 10 10, 10 0, 0 0))
dtype: geometry
```

# geopandas.GeoSeries.transform

`GeoSeries.transform(transformation, include_z=False)`

[\[source\]](#)

Returns a `GeoSeries` with the transformation function applied to the geometry coordinates.

## Parameters:

`transformation : Callable`

A function that transforms a (N, 2) or (N, 3) ndarray of float64 to another (N,2) or (N, 3) ndarray of float64

`include_z : bool, default False`

If True include the third dimension in the coordinates array that is passed to the `transformation` function. If a geometry has no third dimension, the z-coordinates passed to the function will be NaN.

## Returns:

`GeoSeries`

## Examples

```
>>> from shapely import Point, Polygon
>>> s = geopandas.GeoSeries([Point(0, 0)])
>>> s.transform(lambda x: x + 1)
0 POINT (1 1)
dtype: geometry
```

```
>>> s = geopandas.GeoSeries([Polygon([(0, 0), (1, 1), (0, 1)])])
>>> s.transform(lambda x: x * [2, 3])
0 POLYGON ((0 0, 2 3, 0 3, 0 0))
dtype: geometry
```

By default the third dimension is ignored and you need explicitly include it:

```
>>> s = geopandas.GeoSeries([Point(0, 0, 0)])
>>> s.transform(lambda x: x + 1, include_z=True)
0 POINT Z (1 1 1)
dtype: geometry
```

# geopandas.GeoSeries.affine\_transform

`GeoSeries.affine_transform(matrix)`

[\[source\]](#)

Return a `GeoSeries` with translated geometries.

See [http://shapely.readthedocs.io/en/stable/manual.html#shapely.affinity.affine\\_transform](http://shapely.readthedocs.io/en/stable/manual.html#shapely.affinity.affine_transform) for details.

## Parameters:

`matrix: List or tuple`

6 or 12 items for 2D or 3D transformations respectively.

For 2D affine transformations, the 6 parameter matrix is `[a, b, d, e, xoff, yoff]`

For 3D affine transformations, the 12 parameter matrix is `[a, b, c, d, e, f, g, h, i, xoff, yoff, zoff]`

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(1, 1),
... LineString([(1, -1), (1, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
>>> s
0 POINT (1 1)
1 LINESTRING (1 -1, 1 0)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.affine_transform([2, 3, 2, 4, 5, 2])
0 POINT (10 8)
1 LINESTRING (4 0, 7 4)
2 POLYGON ((8 4, 13 10, 14 12, 8 4))
dtype: geometry
```

# geopandas.GeoSeries.rotate

`GeoSeries.rotate(angle, origin='center', use_radians=False)`

[\[source\]](#)

Returns a `GeoSeries` with rotated geometries.

See <http://shapely.readthedocs.io/en/latest/manual.html#shapely.affinity.rotate> for details.

## Parameters:

### `angle : float`

The angle of rotation can be specified in either degrees (default) or radians by setting `use_radians=True`. Positive angles are counter-clockwise and negative are clockwise rotations.

### `origin : string, Point, or tuple (x, y)`

The point of origin can be a keyword ‘center’ for the bounding box center (default), ‘centroid’ for the geometry’s centroid, a `Point` object or a coordinate tuple  $(x, y)$ .

### `use_radians : boolean`

Whether to interpret the angle of rotation as degrees or radians

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(1, 1),
... LineString([(1, -1), (1, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
>>> s
0 POINT (1 1)
1 LINESTRING (1 -1, 1 0)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.rotate(90)
0 POINT (1 1)
1 LINESTRING (1.5 -0.5, 0.5 -0.5)
2 POLYGON ((4.5 -0.5, 3.5 0.5, 2.5 -0.5, 4.5 -0.5))
dtype: geometry
```

```
>>> s.rotate(90, origin=(0, 0))
0 POINT (-1 1)
1 LINESTRING (1 1, 0 1)
2 POLYGON ((1 3, 0 4, -1 3, 1 3))
dtype: geometry
```

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.scale

`GeoSeries.scale(xfact=1.0, yfact=1.0, zfact=1.0, origin='center')`

[\[source\]](#)

Returns a `GeoSeries` with scaled geometries.

The geometries can be scaled by different factors along each dimension. Negative scale factors will mirror or reflect coordinates.

See <http://shapely.readthedocs.io/en/latest/manual.html#shapely.affinity.scale> for details.

## Parameters:

`xfact, yfact, zfact : float, float, float`

Scaling factors for the x, y, and z dimensions respectively.

`origin : string, Point, or tuple`

The point of origin can be a keyword ‘center’ for the 2D bounding box center (default), ‘centroid’ for the geometry’s 2D centroid, a Point object or a coordinate tuple (x, y, z).

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(1, 1),
... LineString([(1, -1), (1, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
>>> s
0 POINT (1 1)
1 LINESTRING (1 -1, 1 0)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.scale(2, 3)
0 POINT (1 1)
1 LINESTRING (1 -2, 1 1)
2 POLYGON ((2.5 -3, 4.5 0, 2.5 3, 2.5 -3))
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.skew

`GeoSeries.skew(xs=0.0, ys=0.0, origin='center', use_radians=False)`

[\[source\]](#)

Returns a `GeoSeries` with skewed geometries.

The geometries are sheared by angles along the x and y dimensions.

See <http://shapely.readthedocs.io/en/latest/manual.html#shapely.affinity.skew> for details.

## Parameters:

**xs, ys : float, float**

The shear angle(s) for the x and y axes respectively. These can be specified in either degrees (default) or radians by setting `use_radians=True`.

**origin : string, Point, or tuple (x, y)**

The point of origin can be a keyword ‘center’ for the bounding box center (default), ‘centroid’ for the geometry’s centroid, a `Point` object or a coordinate tuple (x, y).

**use\_radians : boolean**

Whether to interpret the shear angle(s) as degrees or radians

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(1, 1),
... LineString([(1, -1), (1, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
>>> s
0 POINT (1 1)
1 LINESTRING (1 -1, 1 0)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.skew(45, 30)
0 POINT (1 1)
1 LINESTRING (0.5 -1, 1.5 0)
2 POLYGON ((2 -1.28868, 4 0.28868, 4 0.71132, 2 ...)
dtype: geometry
```

```
>>> s.skew(45, 30, origin=(0, 0))
```

```
0 POINT (2 1.57735)
1 LINESTRING (0 -0.42265, 1 0.57735)
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.translate

`GeoSeries.translate(xoff=0.0, yoff=0.0, zoff=0.0)`

[\[source\]](#)

Returns a `GeoSeries` with translated geometries.

See <http://shapely.readthedocs.io/en/latest/manual.html#shapely.affinity.translate> for details.

## Parameters:

`xoff, yoff, zoff : float, float, float`

Amount of offset along each dimension. xoff, yoff, and zoff for translation along the x, y, and z dimensions respectively.

## Examples

```
>>> from shapely.geometry import Point, LineString, Polygon
>>> s = geopandas.GeoSeries(
... [
... Point(1, 1),
... LineString([(1, -1), (1, 0)]),
... Polygon([(3, -1), (4, 0), (3, 1)]),
...]
...)
>>> s
0 POINT (1 1)
1 LINESTRING (1 -1, 1 0)
2 POLYGON ((3 -1, 4 0, 3 1, 3 -1))
dtype: geometry
```

```
>>> s.translate(2, 3)
0 POINT (3 4)
1 LINESTRING (3 2, 3 3)
2 POLYGON ((5 2, 6 3, 5 4, 5 2))
dtype: geometry
```

# geopandas.GeoSeries.interpolate

`GeoSeries.interpolate(distance, normalized=False)`

[\[source\]](#)

Return a point at the specified distance along each geometry

## Parameters:

**distance : float or Series of floats**

Distance(s) along the geometries at which a point should be returned. If np.array or pd.Series are used then it must have same length as the GeoSeries.

**normalized : boolean**

If normalized is True, distance will be interpreted as a fraction of the geometric object's length.

## Examples

```
>>> from shapely.geometry import LineString, Point
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (2, 0), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
...],
...)
>>> s
0 LINESTRING (0 0, 2 0, 0 2)
1 LINESTRING (0 0, 2 2)
2 LINESTRING (2 0, 0 2)
dtype: geometry
```

```
>>> s.interpolate(1)
0 POINT (1 0)
1 POINT (0.70711 0.70711)
2 POINT (1.29289 0.70711)
dtype: geometry
```

```
>>> s.interpolate([1, 2, 3])
0 POINT (1 0)
1 POINT (1.41421 1.41421)
2 POINT (0 2)
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.line\_merge

`GeoSeries.line_merge(directed=False)`

[\[source\]](#)

Returns (Multi)LineStrings formed by combining the lines in a MultiLineString.

Lines are joined together at their endpoints in case two lines are intersecting. Lines are not joined when 3 or more lines are intersecting at the endpoints. Line elements that cannot be joined are kept as is in the resulting MultiLineString.

The direction of each merged LineString will be that of the majority of the LineStrings from which it was derived. Except if `directed=True` is specified, then the operation will not change the order of points within lines and so only lines which can be joined with no change in direction are merged.

Non-linear geometries result in an empty GeometryCollection.

## Parameters:

`directed : bool, default False`

Only combine lines if possible without changing point order. Requires GEOS >= 3.11.0

## Returns:

`GeoSeries`

## Examples

```
>>> from shapely.geometry import MultiLineString, Point
>>> s = geopandas.GeoSeries(
... [
... MultiLineString([(0, 2), (0, 10)], [(0, 10), (5, 10)]),
... MultiLineString([(0, 2), (0, 10)], [(0, 11), (5, 10)]),
... MultiLineString(),
... MultiLineString([(0, 0), (1, 0)], [(0, 0), (3, 0)]),
... Point(0, 0),
...]
...)
>>> s
0 MULTILINESTRING ((0 2, 0 10), (0 10, 5 10))
1 MULTILINESTRING ((0 2, 0 10), (0 11, 5 10))
2 MULTILINESTRING EMPTY
3 MULTILINESTRING ((0 0, 1 0), (0 0, 3 0))
4 POINT (0 0)
dtype: geometry
```

```
>>> s.line_merge()
0 LINESTRING (0 2, 0 10, 5 10)
1 MULTILINESTRING ((0 2, 0 10), (0 11, 5 10))
2 GEOMETRYCOLLECTION EMPTY
3 LINESTRING (1 0, 0 0, 3 0)
4 GEOMETRYCOLLECTION EMPTY
dtype: geometry
```

With `directed=True`, you can avoid changing the order of points within lines and merge only lines where no change of direction is required:

```
>>> s.line_merge(directed=True)
0 LINESTRING (0 2, 0 10, 5 10)
1 MULTILINESTRING ((0 2, 0 10), (0 11, 5 10))
2 GEOMETRYCOLLECTION EMPTY
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

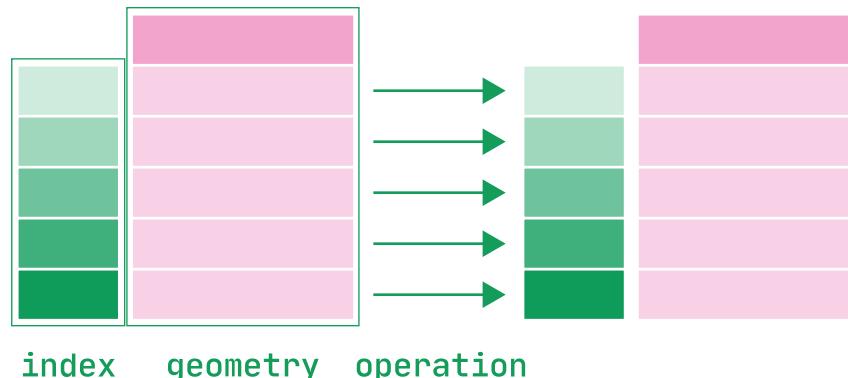
# geopandas.GeoSeries.project

`GeoSeries.project(other, normalized=False, align=None)`

[\[source\]](#)

Return the distance along each geometry nearest to *other*

The operation works on a 1-to-1 row-wise manner:



The project method is the inverse of interpolate.

In shapely, this is equal to [line\\_locate\\_point](#).

## Parameters:

**other : *BaseGeometry or GeoSeries***

The *other* geometry to computed projected point from.

**normalized : *boolean***

If normalized is True, return the distance normalized to the length of the object.

**align : *bool | None (default None)***

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

**Series**

### See also

[GeoSeries.interpolate](#)

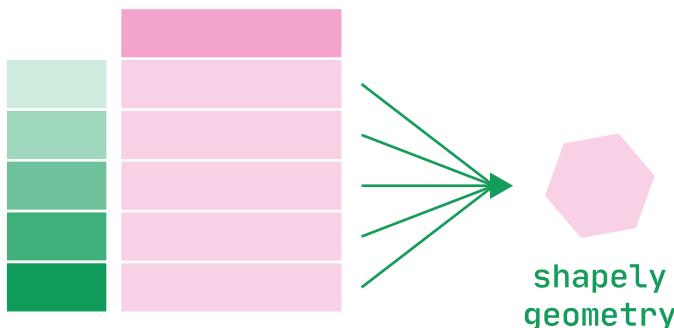
## Examples

```
>>> from shapely.geometry import LineString, Point
>>> s = geopandas.GeoSeries(
... [
... LineString([(0, 0), (2, 0), (0, 2)]),
... LineString([(0, 0), (2, 2)]),
... LineString([(2, 0), (0, 2)]),
...],
...)
>>> s2 = geopandas.GeoSeries(
... [
... Point(1, 0),
... Point(1, 0),
... Point(2, 1),
...],
... index=range(1, 4),
...)
```

```
>>> s
0 LINESTRING (0 0, 2 0, 0 2)
1 LINESTRING (0 0, 2 2)
2 LINESTRING (2 0, 0 2)
dtype: geometry
```

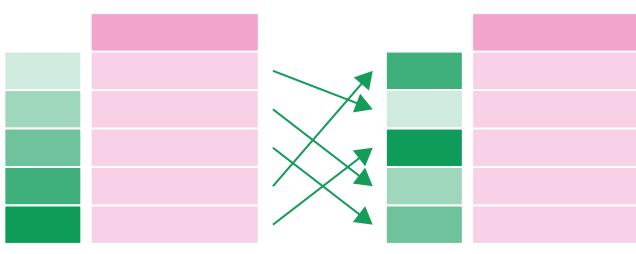
```
>>> s2
1 POINT (1 0)
2 POINT (1 0)
3 POINT (2 1)
dtype: geometry
```

We can project each geometry on a single shapely geometry:

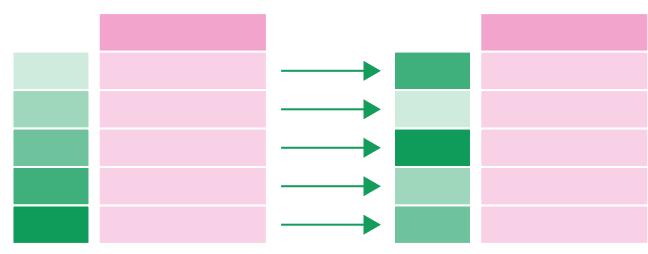


```
>>> s.project(Point(1, 0))
0 1.000000
1 0.707107
2 0.707107
dtype: float64
```

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices. We can either align both GeoSeries based on index values and project elements with the same index using `align=True` or ignore index and project elements based on their matching order using `align=False`:



align=True



align=False

```
>>> s.project(s2, align=True)
0 NaN
1 0.707107
2 0.707107
3 NaN
dtype: float64
```

```
>>> s.project(s2, align=False)
0 1.000000
1 0.707107
2 0.707107
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.shared\_paths

`GeoSeries.shared_paths(other, align=None)`

[\[source\]](#)

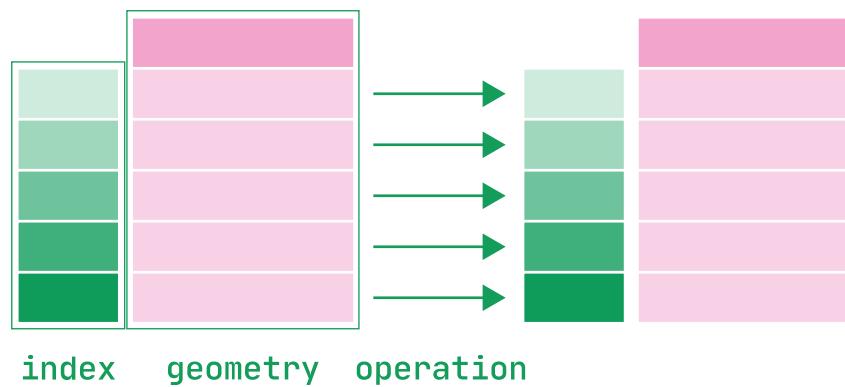
Returns the shared paths between two geometries.

Geometries within the GeoSeries should be only (Multi)LineStrings or LinearRings. A GeoSeries of GeometryCollections is returned with two elements in each GeometryCollection. The first element is a MultiLineString containing shared paths with the same direction for both inputs. The second element is a MultiLineString containing shared paths with the opposite direction for the two inputs.

You can extract individual geometries of the resulting GeometryCollection using the

[`GeoSeries.get\_geometry\(\)`](#) method.

The operation works on a 1-to-1 row-wise manner:



## Parameters:

`other : Geoseries or geometric object`

The Geoseries (elementwise) or geometric object to find the shared paths with. Has to contain only (Multi)LineString or LinearRing geometry types.

`align : bool | None (default None)`

If True, automatically aligns GeoSeries based on their indices. If False, the order of elements is preserved. None defaults to True.

## Returns:

`GeoSeries`

## See also

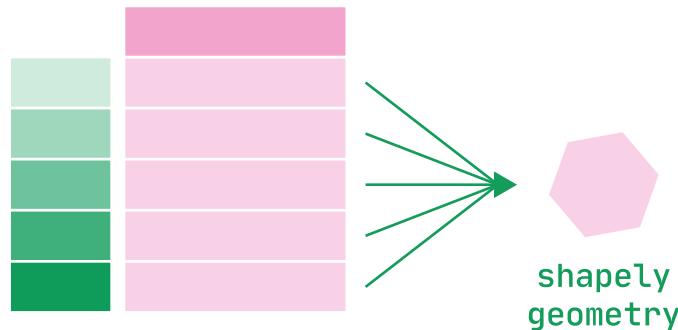
[`GeoSeries.get\_geometry`](#)

## Examples

```
>>> from shapely.geometry import LineString, MultiLineString
>>> s = geopandas.GeoSeries([
... [
... LineString([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(1, 0), (2, 0), (2, 1), (1, 1), (1, 0)]),
... MultiLineString([(1, 0), (2, 0)], [(2, 1), (1, 1), (1, 0)]),
...],
...)
>>> s
0 LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)
1 LINESTRING (1 0, 2 0, 2 1, 1 1, 1 0)
2 MULTILINESTRING ((1 0, 2 0), (2 1, 1 1, 1 0))
dtype: geometry
```

>>>

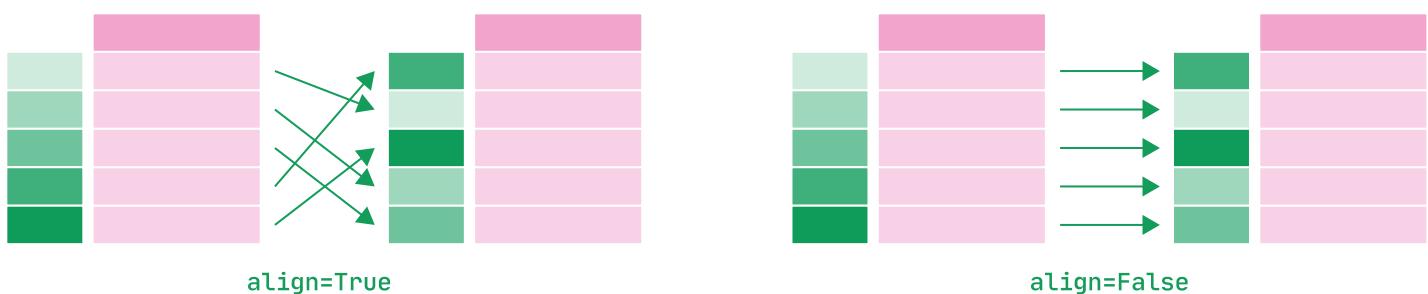
We can find the shared paths between each geometry and a single shapely geometry:



```
>>> l = LineString([(1, 1), (0, 1)])
>>> s.shared_paths(l)
0 GEOMETRYCOLLECTION (MULTILINESTRING ((1 1, 0 1...
1 GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MUL...
2 GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MUL...
dtype: geometry
```

>>>

We can also check two GeoSeries against each other, row by row. The GeoSeries above have different indices than the one below. We can either align both GeoSeries based on index values and compare elements with the same index using `align=True` or ignore index and compare elements based on their matching order using `align=False`:



```
>>> s2 = geopandas.GeoSeries(
... [
... LineString([(1, 1), (0, 1)]),
... LineString([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]),
... LineString([(1, 0), (2, 0), (2, 1), (1, 1), (1, 0)]),
...],
... index=[1, 2, 3]
...)
```

```
>>> s.shared_paths(s2, align=True)
0 None
1 GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MUL...
2 GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MUL...
3 None
dtype: geometry
>>>
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.build\_area

`GeoSeries.build_area(node=True)`

[\[source\]](#)

Creates an areal geometry formed by the constituent linework.

Builds areas from the GeoSeries that contain linework which represents the edges of a planar graph. Any geometry type may be provided as input; only the constituent lines and rings will be used to create the output polygons. All geometries within the GeoSeries are considered together and the resulting polygons therefore do not map 1:1 to input geometries.

This function converts inner rings into holes. To turn inner rings into polygons as well, use polygonize.

Unless you know that the input GeoSeries represents a planar graph with a clean topology (e.g. there is a node on both lines where they intersect), it is recommended to use `node=True` which performs noding prior to building areal geometry. Using `node=False` will provide performance benefits but may result in incorrect polygons if the input is not of the proper topology.

If the input linework crosses, this function may produce invalid polygons. Use

[`GeoSeries.make\_valid\(\)`](#) to ensure valid geometries.

## Parameters:

`node : bool, default True`

Perform noding prior to building the areas, by default True.

## Returns:

`GeoSeries`

GeoSeries with polygons

## Examples

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.explode

`GeoSeries.explode(ignore_index=False, index_parts=False)`

[\[source\]](#)

Explode multi-part geometries into multiple single geometries.

Single rows can become multiple rows. This is analogous to PostGIS's ST\_Dump(). The 'path' index is the second level of the returned MultiIndex

## Parameters:

`ignore_index : bool, default False`

If True, the resulting index will be labelled 0, 1, ..., n - 1, ignoring `index_parts`.

`index_parts : boolean, default False`

If True, the resulting index will be a multi-index (original index with an additional level indicating the multiple geometries: a new zero-based index for each single part geometry per multi-part geometry).

## Returns:

A GeoSeries with a MultiIndex. The levels of the MultiIndex are the original index and a zero-based integer index that counts the number of single geometries within a multi-part geometry.

## See also

[GeoDataFrame.explode](#)

## Examples

```
>>> from shapely.geometry import MultiPoint
>>> s = geopandas.GeoSeries(
... [MultiPoint([(0, 0), (1, 1)]), MultiPoint([(2, 2), (3, 3), (4, 4)])]
...)
>>> s
0 MULTIPOINT ((0 0), (1 1))
1 MULTIPOINT ((2 2), (3 3), (4 4))
dtype: geometry
```

```
>>> s.explode(index_parts=True)
0 0 POINT (0 0)
1 1 POINT (1 1)
1 0 POINT (2 2)
1 1 POINT (3 3)
2 2 POINT (4 4)
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.intersection\_all

## GeoSeries.intersection\_all()

[\[source\]](#)

Returns a geometry containing the intersection of all geometries in the `GeoSeries`.

This method ignores `None` values when other geometries are present. If all elements of the `GeoSeries` are `None`, an empty `GeometryCollection` is returned.

## Examples

```
>>> from shapely.geometry import box
>>> s = geopandas.GeoSeries(
... [box(0, 0, 2, 2), box(1, 1, 3, 3), box(0, 0, 1.5, 1.5)])
...)
>>> s
0 POLYGON ((2 0, 2 2, 0 2, 0 0, 2 0))
1 POLYGON ((3 1, 3 3, 1 3, 1 1, 3 1))
2 POLYGON ((1.5 0, 1.5 1.5, 0 1.5, 0 0, 1.5 0))
dtype: geometry
```

```
>>> s.intersection_all()
<POLYGON ((1 1, 1 1.5, 1.5 1.5, 1.5 1, 1 1))>
```

# geopandas.GeoSeries.polygonize

`GeoSeries.polygonize(node=True, full=False)`

[\[source\]](#)

Creates polygons formed from the linework of a GeoSeries.

Polygonizes the GeoSeries that contain linework which represents the edges of a planar graph. Any geometry type may be provided as input; only the constituent lines and rings will be used to create the output polygons.

Lines or rings that when combined do not completely close a polygon will be ignored. Duplicate segments are ignored.

Unless you know that the input GeoSeries represents a planar graph with a clean topology (e.g. there is a node on both lines where they intersect), it is recommended to use `node=True` which performs noding prior to polygonization. Using `node=False` will provide performance benefits but may result in incorrect polygons if the input is not of the proper topology.

When `full=True`, the return value is a 4-tuple containing output polygons, along with lines which could not be converted to polygons. The return value consists of 4 elements of varying lengths:

- GeoSeries of the valid polygons (same as with `full=False`)
- GeoSeries of cut edges: edges connected on both ends but not part of polygonal output
- GeoSeries of dangles: edges connected on one end but not part of polygonal output
- GeoSeries of invalid rings: polygons that are formed but are not valid (bowties, etc)

## Parameters:

**node : bool, default True**

Perform noding prior to polygonization, by default True.

**full : bool, default False**

Return the full output composed of a tuple of GeoSeries, by default False.

## Returns:

`GeoSeries | tuple(GeoSeries, GeoSeries, GeoSeries, GeoSeries)`

GeoSeries with the polygons or a tuple of four GeoSeries as `(polygons, cuts, dangles, invalid)`

## Examples

```
>>> from shapely.geometry import LineString
>>> s = geopandas.GeoSeries([
... LineString([(0, 0), (1, 1)]),
... LineString([(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)]),
... LineString([(0.5, 0.2), (0.5, 0.8)]),
...])
>>> s.polygonize()
0 POLYGON ((0 0, 0.5 0.5, 1 1, 1 0, 0 0))
1 POLYGON ((0.5 0.5, 0 0, 0 1, 1 1, 0.5 0.5))
Name: polygons, dtype: geometry
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.union\_all

`GeoSeries.union_all(method='unary')`

[\[source\]](#)

Returns a geometry containing the union of all geometries in the `GeoSeries`.

By default, the unary union algorithm is used. If the geometries are non-overlapping (forming a coverage), GeoPandas can use a significantly faster algorithm to perform the union using the `method="coverage"` option.

## Parameters:

`method : str (default "unary")`

The method to use for the union. Options are:

- `"unary"`: use the unary union algorithm. This option is the most robust but can be slow for large numbers of geometries (default).
- `"coverage"`: use the coverage union algorithm. This option is optimized for non-overlapping polygons and can be significantly faster than the unary union algorithm. However, it can produce invalid geometries if the polygons overlap.

## Examples

```
>>> from shapely.geometry import box
>>> s = geopandas.GeoSeries([box(0, 0, 1, 1), box(0, 0, 2, 2)])
>>> s
0 POLYGON ((1 0, 1 1, 0 1, 0 0, 1 0))
1 POLYGON ((2 0, 2 2, 0 2, 0 0, 2 0))
dtype: geometry
```

```
>>> s.union_all()
<POLYGON ((0 1, 0 2, 2 2, 2 0, 1 0, 0 0, 0 1))>
```



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoSeries.from\_arrow

*classmethod* `GeoSeries.from_arrow(arr, **kwargs)`

[\[source\]](#)

Construct a GeoSeries from a Arrow array object with a GeoArrow extension type.

See <https://geoarrow.org/> for details on the GeoArrow specification.

This function accepts any Arrow array object implementing the [Arrow PyCapsule Protocol](#) (i.e. having an `__arrow_c_array__` method).

 **Added in version 1.0.**

## Parameters:

`arr : pyarrow.Array, Arrow array`

Any array object implementing the Arrow PyCapsule Protocol (i.e. has an `__arrow_c_array__` or `__arrow_c_stream__` method). The type of the array should be one of the geoarrow geometry types.

`**kwargs`

Other parameters passed to the GeoSeries constructor.

## Returns:

`GeoSeries`

# geopandas.GeoSeries.from\_file

*classmethod* `GeoSeries.from_file(filename, **kwargs)`

[\[source\]](#)

Alternate constructor to create a `GeoSeries` from a file.

Can load a `GeoSeries` from a file from any format recognized by `pyogrio`. See <http://pyogrio.readthedocs.io/> for details. From a file with attributes loads only geometry column. Note that to do that, GeoPandas first loads the whole GeoDataFrame.

## Parameters:

`filename : str`

File path or file handle to read from. Depending on which kwargs are included, the content of filename may vary. See [`pyogrio.read\_dataframe\(\)`](#) for usage details.

`kwargs : key-word arguments`

These arguments are passed to [`pyogrio.read\_dataframe\(\)`](#), and can be used to access multi-layer data, data stored within archives (zip files), etc.

## See also

[`read\_file`](#)

read file to GeoDataFrame

## Examples

```
>>> import geodatasets
>>> path = geodatasets.get_path('nybb')
>>> s = geopandas.GeoSeries.from_file(path)
>>> s
0 MULTIPOLYGON (((970217.022 145643.332, 970227....
1 MULTIPOLYGON (((1029606.077 156073.814, 102957...
2 MULTIPOLYGON (((1021176.479 151374.797, 102100...
3 MULTIPOLYGON (((981219.056 188655.316, 980940....
4 MULTIPOLYGON (((1012821.806 229228.265, 101278...
Name: geometry, dtype: geometry
```

# geopandas.GeoSeries.from\_wkb

*classmethod* `GeoSeries.from_wkb(data, index=None, crs=None, on_invalid='raise', **kwargs)`

[\[source\]](#)

Alternate constructor to create a `GeoSeries` from a list or array of WKB objects

## Parameters:

### `data : array-like or Series`

Series, list or array of WKB objects

### `index : array-like or Index`

The index for the GeoSeries.

### `crs : value, optional`

Coordinate Reference System of the geometry objects. Can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

### `on_invalid: {"raise", "warn", "ignore"}, default "raise"`

- raise: an exception will be raised if a WKB input geometry is invalid.
- warn: a warning will be raised and invalid WKB geometries will be returned as None.
- ignore: invalid WKB geometries will be returned as None without a warning.

## kwargs

Additional arguments passed to the Series constructor, e.g. `name`.

## Returns:

### `GeoSeries`

## See also

[GeoSeries.from\\_wkt](#)

# geopandas.GeoSeries.from\_wkt

*classmethod* `GeoSeries.from_wkt(data, index=None, crs=None, on_invalid='raise', **kwargs)`

[\[source\]](#)

Alternate constructor to create a `GeoSeries` from a list or array of WKT objects

## Parameters:

`data : array-like, Series`

Series, list, or array of WKT objects

`index : array-like or Index`

The index for the GeoSeries.

`crs : value, optional`

Coordinate Reference System of the geometry objects. Can be anything accepted by

[`pyproj.CRS.from\_user\_input\(\)`](#), such as an authority string (eg “EPSG:4326”) or a WKT string.

`on_invalid : {"raise", "warn", "ignore"}, default "raise"`

- raise: an exception will be raised if a WKT input geometry is invalid.
- warn: a warning will be raised and invalid WKT geometries will be returned as `None`.
- ignore: invalid WKT geometries will be returned as `None` without a warning.

`kwargs`

Additional arguments passed to the Series constructor, e.g. `name`.

## Returns:

`GeoSeries`

## See also

[`GeoSeries.from\_wkb`](#)

## Examples

```
>>> wkts = [
... 'POINT (1 1)',
... 'POINT (2 2)',
... 'POINT (3 3)'].
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.from\_xy

**classmethod** `GeoSeries.from_xy(x, y, z=None, index=None, crs=None, **kwargs)`

Alternate constructor to create a `GeoSeries` of Point geometries from lists or arrays of x, y, ([\[source\]](#)) z) coordinates

In case of geographic coordinates, it is assumed that longitude is captured by `x` coordinates and latitude by `y`.

## Parameters:

`x, y, z : iterable`

`index : array-like or Index, optional`

The index for the GeoSeries. If not given and all coordinate inputs are Series with an equal index, that index is used.

`crs : value, optional`

Coordinate Reference System of the geometry objects. Can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

`**kwargs`

Additional arguments passed to the Series constructor, e.g. `name`.

## Returns:

`GeoSeries`

## See also

[GeoSeries.from\\_wkt](#)

[points\\_from\\_xy](#)

## Examples

```
>>> x = [2.5, 5, -3.0]
>>> y = [0.5, 1, 1.5]
>>> s = geopandas.GeoSeries.from_xy(x, y, crs="EPSG:4326")
>>> s
0 POINT (2.5 0.5)
1 POINT (5 1)
2 POINT (-3 1.5)
dtype: geometry
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.to\_arrow

`GeoSeries.to_arrow(geometries_encoding='WKB', interleaved=True, include_z=None)`

[\[source\]](#)

Encode a GeoSeries to GeoArrow format.

See <https://geoarrow.org/> for details on the GeoArrow specification.

This function returns a generic Arrow array object implementing the [Arrow PyCapsule Protocol](#) (i.e. having an `__arrow_c_array__` method). This object can then be consumed by your Arrow implementation of choice that supports this protocol.

 **Added in version 1.0.**

## Parameters:

**geometries\_encoding : {‘WKB’, ‘geoarrow’}, default ‘WKB’**

The GeoArrow encoding to use for the data conversion.

**interleaved : bool, default True**

Only relevant for ‘geoarrow’ encoding. If True, the geometries’ coordinates are interleaved in a single fixed size list array. If False, the coordinates are stored as separate arrays in a struct type.

**include\_z : bool, default None**

Only relevant for ‘geoarrow’ encoding (for WKB, the dimensionality of the individual geometries is preserved). If False, return 2D geometries. If True, include the third dimension in the output (if a geometry has no third dimension, the z-coordinates will be NaN). By default, will infer the dimensionality from the input geometries. Note that this inference can be unreliable with empty geometries (for a guaranteed result, it is recommended to specify the keyword).

## Returns:

**GeoArrowArray**

A generic Arrow array object with geometry data encoded to GeoArrow.

## Examples

```
>>> from shapely.geometry import Point
>>> gser = geopandas.GeoSeries([Point(1, 2), Point(2, 1)])
>>> gser
0 POINT (1 2)
1 POINT (2 1)
dtype: geometry
```

```
>>> arrow_array = gser.to_arrow()
>>> arrow_array
<geopandas.io._geoarrow.GeoArrowArray object at ...>
```

The returned array object needs to be consumed by a library implementing the Arrow PyCapsule Protocol. For example, wrapping the data as a pyarrow.Array (requires pyarrow >= 14.0):

```
>>> import pyarrow as pa
>>> array = pa.array(arrow_array)
>>> array
<pyarrow.lib.BinaryArray object at ...>
[
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.to\_file

`GeoSeries.to_file(filename, driver=None, index=None, **kwargs)`

[\[source\]](#)

Write the `GeoSeries` to a file.

By default, an ESRI shapefile is written, but any OGR data source supported by Pyogrio or Fiona can be written.

## Parameters:

### `filename : string`

File path or file handle to write to. The path may specify a GDAL VSI scheme.

### `driver : string, default None`

The OGR format driver used to write the vector file. If not specified, it attempts to infer it from the file extension. If no extension is specified, it saves ESRI Shapefile to a folder.

### `index : bool, default None`

If True, write index into one or more columns (for MultiIndex). Default None writes the index into one or more columns only if the index is named, is a MultiIndex, or has a non-integer data type. If False, no index is written.

 **Added in version 0.7:** Previously the index was not written.

### `mode : string, default ‘w’`

The write mode, ‘w’ to overwrite the existing file and ‘a’ to append. Not all drivers support appending. The drivers that support appending are listed in `fiona.supported_drivers` or

 [Toblerity/Fiona](#)

### `crs : pyproj.CRS, default None`

If specified, the CRS is passed to Fiona to better control how the file is written. If None, GeoPandas will determine the crs based on crs df attribute. The value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string. The keyword is not supported for the “pyogrio” engine.

### `engine : str, “pyogrio” or “fiona”`

The underlying library that is used to write the file. Currently, the supported options are “pyogrio” and “fiona”. Defaults to “pyogrio” if installed, otherwise tries “fiona”.

### `**kwargs`

Keyword args to be passed to the engine, and can be used to write to multi-layer data, store data within archives (zip files), etc. In case of the “pyogrio” engine, the keyword arguments are passed to `pyogrio.write_dataframe`. In case of the “fiona” engine, the keyword arguments

are passed to `fiona.open`. For more information on possible keywords, type: `import pyogrio; help(pyogrio.write_dataframe)`.

## See also

[`GeoDataFrame.to\_file`](#)

write GeoDataFrame to file

[`read\_file`](#)

read file to GeoDataFrame

## Examples

```
>>> s.to_file('series.shp')
```

```
>>> s.to_file('series.gpkg', driver='GPKG', layer='name1')
```

```
>>> s.to_file('series.geojson', driver='GeoJSON')
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.to\_json

`GeoSeries.to_json(show_bbox=True, drop_id=False, to_wgs84=False, **kwargs)`

Returns a GeoJSON string representation of the GeoSeries.

[\[source\]](#)

## Parameters:

**show\_bbox : bool, optional, default: True**

Include bbox (bounds) in the geojson

**drop\_id : bool, default: False**

Whether to retain the index of the GeoSeries as the id property in the generated GeoJSON.

Default is False, but may want True if the index is just arbitrary row numbers.

**to\_wgs84: bool, optional, default: False**

If the CRS is set on the active geometry column it is exported as WGS84 (EPSG:4326) to

meet the [2016 GeoJSON specification](#). Set to True to force re-projection and set to False to

ignore CRS. False by default.

\*kwargs\* that will be passed to json.dumps().

## Returns:

JSON string

## See also

[GeoSeries.to\\_file](#)

write GeoSeries to file

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)])
>>> s
0 POINT (1 1)
1 POINT (2 2)
2 POINT (3 3)
dtype: geometry
```

```
>>> s.to_json()
'{"type": "FeatureCollection", "features": [{"id": "0", "type": "Feature", "propertie
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.to\_wkb

`GeoSeries.to_wkb(hex=False, **kwargs)`

[\[source\]](#)

Convert GeoSeries geometries to WKB

## Parameters:

`hex : bool`

If true, export the WKB as a hexadecimal string. The default is to return a binary bytes object.

`kwargs`

Additional keyword args will be passed to [`shapely.to\_wkb\(\)`](#).

## Returns:

`Series`

WKB representations of the geometries

## See also

[`GeoSeries.to\_wkt`](#)

# geopandas.GeoSeries.to\_wkt

`GeoSeries.to_wkt( **kwargs )`

[\[source\]](#)

Convert GeoSeries geometries to WKT

## Parameters:

`kwargs`

Keyword args will be passed to [`shapely.to\_wkt\(\)`](#).

## Returns:

`Series`

WKT representations of the geometries

## See also

[`GeoSeries.to\_wkb`](#)

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)])
>>> s
0 POINT (1 1)
1 POINT (2 2)
2 POINT (3 3)
dtype: geometry
```

```
>>> s.to_wkt()
0 POINT (1 1)
1 POINT (2 2)
2 POINT (3 3)
dtype: object
```

# geopandas.GeoSeries.crs

*property* `GeoSeries.crs`

[\[source\]](#)

The Coordinate Reference System (CRS) represented as a `pyproj.CRS` object.

Returns None if the CRS is not set, and to set the value it :getter: Returns a `pyproj.CRS` or None.

When setting, the value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

## See also

[`GeoSeries.set\_crs`](#)

assign CRS

[`GeoSeries.to\_crs`](#)

re-project to another CRS

## Examples

```
>>> s.crs
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

# geopandas.GeoSeries.set\_crs

`GeoSeries.set_crs(**kwargs)`

[\[source\]](#)

# geopandas.GeoSeries.to\_crs

`GeoSeries.to_crs(crs=None, epsg=None)`

[\[source\]](#)

Returns a `GeoSeries` with all geometries transformed to a new coordinate reference system.

Transform all geometries in a GeoSeries to a different coordinate reference system. The `crs` attribute on the current GeoSeries must be set. Either `crs` or `epsg` may be specified for output.

This method will transform all points in all objects. It has no notion of 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.

## Parameters:

`crs : pyproj.CRS, optional if epsg is specified`

The value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

`epsg : int, optional if crs is specified`

EPSG code specifying output projection.

## Returns:

`GeoSeries`

## See also

[`GeoSeries.set\_crs`](#)

assign CRS

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)], crs=4326)
>>> s
0 POINT (1 1)
1 POINT (2 2)
2 POINT (3 3)
dtype: geometry
>>> s.crs
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
>>> s = s.to_crs(3857)
>>> s
0 POINT (111319.491 111325.143)
1 POINT (222638.982 222684.209)
2 POINT (333958.472 334111.171)
dtype: geometry
>>> s.crs
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.estimate\_utm\_crs

`GeoSeries.estimate_utm_crs(datum_name='WGS 84')`

[\[source\]](#)

Returns the estimated UTM CRS based on the bounds of the dataset.

**!** *Added in version 0.9.*

## Parameters:

`datum_name : str, optional`

The name of the datum to use in the query. Default is WGS 84.

## Returns:

`pypj.CRS`

## Examples

```
>>> import geodatasets
>>> df = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health")
...)
>>> df.geometry.estimate_utm_crs()
<Derived Projected CRS: EPSG:32616>
Name: WGS 84 / UTM zone 16N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Between 90°W and 84°W, northern hemisphere between equator and 84°N, ...
- bounds: (-90.0, 0.0, -84.0, 84.0)
Coordinate Operation:
- name: UTM zone 16N
- method: Transverse Mercator
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

# geopandas.GeoSeries.fillna

`GeoSeries.fillna(value=None, inplace=False, limit=None, **kwargs)`

[\[source\]](#)

Fill NA values with geometry (or geometries).

## Parameters:

`value : shapely geometry or GeoSeries, default None`

If `None` is passed, NA values will be filled with GEOMETRYCOLLECTION EMPTY. If a shapely geometry object is passed, it will be used to fill all missing values. If a `GeoSeries` or `GeometryArray` are passed, missing values will be filled based on the corresponding index locations. If `pd.NA` or `np.nan` are passed, values will be filled with `None` (not GEOMETRYCOLLECTION EMPTY).

`limit : int, default None`

This is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not `None`.

## Returns:

`GeoSeries`

## See also

[`GeoSeries.isna`](#)

detect missing values

## Examples

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [
... Polygon([(0, 0), (1, 1), (0, 1)]),
... None,
... Polygon([(0, 0), (-1, 1), (0, -1)]),
...]
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 None
2 POLYGON ((0 0, -1 1, 0 -1, 0 0))
dtype: geometry
```

Filled with an empty polygon.

```
>>> s.fillna()
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 GEOMETRYCOLLECTION EMPTY
2 POLYGON ((0 0, -1 1, 0 -1, 0 0))
dtype: geometry
```

>>>

Filled with a specific polygon.

```
>>> s.fillna(Polygon([(0, 1), (2, 1), (1, 2)]))
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 POLYGON ((0 1, 2 1, 1 2, 0 1))
2 POLYGON ((0 0, -1 1, 0 -1, 0 0))
dtype: geometry
```

>>>

Filled with another GeoSeries.

```
>>> from shapely.geometry import Point
>>> s_fill = geopandas.GeoSeries(
... [
... Point(0, 0),
... Point(1, 1),
... Point(2, 2),
...]
...)
>>> s.fillna(s_fill)
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 POINT (1 1)
2 POLYGON ((0 0, -1 1, 0 -1, 0 0))
dtype: geometry
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.isna

## GeoSeries.isna()

[source]

Detect missing values.

Historically, NA values in a GeoSeries could be represented by empty geometric objects, in addition to standard representations such as None and np.nan. This behaviour is changed in version 0.6.0, and now only actual missing values return True. To detect empty geometries, use [GeoSeries.is\\_empty](#) instead.

### Returns:

A boolean pandas Series of the same size as the GeoSeries,

True where a value is NA.

### See also

[GeoSeries.notna](#)

inverse of isna

[GeoSeries.is\\_empty](#)

detect empty geometries

## Examples

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [Polygon([(0, 0), (1, 1), (0, 1)]), None, Polygon([])])
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 None
2 POLYGON EMPTY
dtype: geometry
```

```
>>> s.isna()
0 False
1 True
2 False
dtype: bool
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.notna

## GeoSeries.notna()

[\[source\]](#)

Detect non-missing values.

Historically, NA values in a GeoSeries could be represented by empty geometric objects, in addition to standard representations such as None and np.nan. This behaviour is changed in version 0.6.0, and now only actual missing values return False. To detect empty geometries, use `~GeoSeries.is_empty` instead.

### Returns:

A boolean pandas Series of the same size as the GeoSeries,

False where a value is NA.

### See also

[GeoSeries.isna](#)

inverse of notna

[GeoSeries.is\\_empty](#)

detect empty geometries

## Examples

```
>>> from shapely.geometry import Polygon
>>> s = geopandas.GeoSeries(
... [Polygon([(0, 0), (1, 1), (0, 1)]), None, Polygon([])])
...)
>>> s
0 POLYGON ((0 0, 1 1, 0 1, 0 0))
1 None
2 POLYGON EMPTY
dtype: geometry
```

```
>>> s.notna()
0 True
1 False
2 True
dtype: bool
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.clip

`GeoSeries.clip(mask, keep_geom_type=False, sort=False)`

[\[source\]](#)

Clip points, lines, or polygon geometries to the mask extent.

Both layers must be in the same Coordinate Reference System (CRS). The GeoSeries will be clipped to the full extent of the `mask` object.

If there are multiple polygons in `mask`, data from the GeoSeries will be clipped to the total boundary of all polygons in `mask`.

## Parameters:

**mask : *GeoDataFrame, GeoSeries, (Multi)Polygon, list-like***

Polygon vector layer used to clip `gdf`. The mask's geometry is dissolved into one geometric feature and intersected with GeoSeries. If the mask is list-like with four elements (`minx, miny, maxx, maxy`), `clip` will use a faster rectangle clipping ([`clip\_by\_rect\(\)`](#)), possibly leading to slightly different results.

**keep\_geom\_type : *boolean, default False***

If True, return only geometries of original type in case of intersection resulting in multiple geometry types or GeometryCollections. If False, return all resulting geometries (potentially mixed-types).

**sort : *boolean, default False***

If True, the order of rows in the clipped GeoSeries will be preserved at small performance cost. If False the order of rows in the clipped GeoSeries will be random.

## Returns:

**GeoSeries**

Vector data (points, lines, polygons) from `gdf` clipped to polygon boundary from `mask`.

## See also

[`clip`](#)

top-level function for `clip`

## Examples

Clip points (grocery stores) with polygons (the Near West Side community):

```
>>> import geodatasets
>>> chicago = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health"))
...)
>>> near_west_side = chicago[chicago["community"] == "NEAR WEST SIDE"]
>>> groceries = geopandas.read_file(
... geodatasets.get_path("geoda.groceries"))
...).to_crs(chicago.crs)
>>> groceries.shape
(148, 8)
```

```
>>> nws_groceries = groceries.geometry.clip(near_west_side)
>>> nws_groceries.shape
(7,)
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoSeries.plot

`GeoSeries.plot(*args, **kwargs)`

[\[source\]](#)

Plot a GeoSeries.

Generate a plot of a GeoSeries geometry with matplotlib.

## Parameters:

**s : Series**

The GeoSeries to be plotted. Currently Polygon, MultiPolygon, LineString, MultiLineString, Point and MultiPoint geometries can be plotted.

**cmap : str (default None)**

The name of a colormap recognized by matplotlib. Any colormap will work, but categorical colormaps are generally recommended. Examples of useful discrete colormaps include:

tab10, tab20, Accent, Dark2, Paired, Pastel1, Set1, Set2

**color : str, np.array, pd.Series, List (default None)**

If specified, all objects will be colored uniformly.

**ax : matplotlib.pyplot.Artist (default None)**

axes on which to draw the plot

**figsize : pair of floats (default None)**

Size of the resulting matplotlib.figure.Figure. If the argument ax is given explicitly, figsize is ignored.

**aspect : 'auto', 'equal', None or float (default 'auto')**

Set aspect of axis. If 'auto', the default aspect for map plots is 'equal'; if however data are not projected (coordinates are long/lat), the aspect is by default set to  $1/\cos(s_y \cdot \pi/180)$  with  $s_y$  the y coordinate of the middle of the GeoSeries (the mean of the y range of bounding box) so that a long/lat square appears square in the middle of the plot. This implies an Equirectangular projection. If None, the aspect of ax won't be changed. It can also be set manually (float) as the ratio of y-unit to x-unit.

**autolim : bool (default True)**

Update axes data limits to contain the new geometries.

**\*\*style\_kwds : dict**

Color options to be passed on to the actual plot function, such as `edgecolor`, `facecolor`, `linewidth`, `markersize`, `alpha`.

## Returns:

`ax : matplotlib axes instance`

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.explore

`GeoSeries.explore(*args, **kwargs)`

[\[source\]](#)

Interactive map based on folium/leaflet.jsInteractive map based on GeoPandas and folium/leaflet.js

Generate an interactive leaflet map based on `GeoSeries`

## Parameters:

**color : str, array-like (default None)**

Named color or a list-like of colors (named or hex).

**m : folium.Map (default None)**

Existing map instance on which to draw the plot.

**tiles : str, xyzservices.TileProvider (default ‘OpenStreetMap Mapnik’)**

Map tileset to use. Can choose from the list supported by folium, query a

`xyzservices.TileProvider` by a name from `xyzservices.providers`, pass

`xyzservices.TileProvider` object or pass custom XYZ URL. The current list of built-in providers (when `xyzservices` is not available):

`["OpenStreetMap", "CartoDB positron", "CartoDB dark_matter"]`

You can pass a custom tileset to Folium by passing a Leaflet-style URL to the tiles

parameter: `http://{s}.yourtiles.com/{z}/{x}/{y}.png`. Be sure to check their terms and conditions and to provide attribution with the `attr` keyword.

**attr : str (default None)**

Map tile attribution; only required if passing custom tile URL.

**highlight : bool (default True)**

Enable highlight functionality when hovering over a geometry.

**width : pixel int or percentage string (default: ‘100%’)**

Width of the folium `Map`. If the argument m is given explicitly, width is ignored.

**height : pixel int or percentage string (default: ‘100%’)**

Height of the folium `Map`. If the argument m is given explicitly, height is ignored.

**control\_scale : bool, (default True)**

Whether to add a control scale on the map.

**marker\_type : str, folium.Circle, folium.CircleMarker, folium.Marker (default None)**

Allowed string options are (‘marker’, ‘circle’, ‘circle\_marker’). Defaults to folium.Marker.

**marker\_kwds: dict (default {})**

Additional keywords to be passed to the selected `marker_type`, e.g.:

**radius : float**

Radius of the circle, in meters (for `‘circle’`) or pixels (for `‘circle_marker’`).

**icon : folium.map.Icon**

the [folium.map.Icon](#) object to use to render the marker.

**draggable : bool (default False)**

Set to True to be able to drag the marker around the map.

**style\_kwds : dict (default {})**

Additional style to be passed to folium [style\\_function](#):

**stroke : bool (default True)**

Whether to draw stroke along the path. Set it to [False](#) to disable borders on polygons or circles.

**color : str**

Stroke color

**weight : int**

Stroke width in pixels

**opacity : float (default 1.0)**

Stroke opacity

**fill : boolean (default True)**

Whether to fill the path with color. Set it to [False](#) to disable filling on polygons or circles.

**fillColor : str**

Fill color. Defaults to the value of the color option

**fillOpacity : float (default 0.5)**

Fill opacity.

**style\_function : callable**

Function mapping a GeoJson Feature to a style [dict](#).

- Style properties [folium.vector\\_layers.path\\_options\(\)](#)
- GeoJson features [GeoSeries.\\_\\_geo\\_interface\\_\\_](#)

e.g.:

```
lambda x: {"color": "red" if x["properties"]["gdp_md_est"] < 10**6
 else "blue"}
```

Plus all supported by [folium.vector\\_layers.path\\_options\(\)](#). See the documentation of [folium.features.GeoJson](#) for details.

**highlight\_kwds : dict (default {})**

Style to be passed to folium highlight\_function. Uses the same keywords as [style\\_kwds](#).

When empty, defaults to [{"fillOpacity": 0.75}](#).

**map\_kwds : dict (default {})**

**\*\*kwargs : dict**

Additional options to be passed on to the folium.

**Returns:**

**m : *folium.folium.Map***

folium [Map](#) instance

# geopandas.GeoSeries.sindex

**property** `GeoSeries.sindex`

[\[source\]](#)

Generate the spatial index

Creates R-tree spatial index based on `shapely.STRtree`.

Note that the spatial index may not be fully initialized until the first use.

## Examples

```
>>> from shapely.geometry import box
>>> s = geopandas.GeoSeries(geopandas.points_from_xy(range(5), range(5)))
>>> s
0 POINT (0 0)
1 POINT (1 1)
2 POINT (2 2)
3 POINT (3 3)
4 POINT (4 4)
dtype: geometry
```

Query the spatial index with a single geometry based on the bounding box:

```
>>> s.sindex.query(box(1, 1, 3, 3))
array([1, 2, 3])
```

Query the spatial index with a single geometry based on the predicate:

```
>>> s.sindex.query(box(1, 1, 3, 3), predicate="contains")
array([2])
```

Query the spatial index with an array of geometries based on the bounding box:

```
>>> s2 = geopandas.GeoSeries([box(1, 1, 3, 3), box(4, 4, 5, 5)])
>>> s2
0 POLYGON ((3 1, 3 3, 1 3, 1 1, 3 1))
1 POLYGON ((5 4, 5 5, 4 5, 4 4, 5 4))
dtype: geometry
```

```
>>> s.sindex.query(s2)
array([[0, 0, 0, 1],
 [1, 2, 3, 4]])
```

Query the spatial index with an array of geometries based on the predicate:

```
>>> s.sindex.query(s2, predicate="contains")
array([[0],
 [2]])
```

>>>

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoSeries.has\_sindex

*property* `GeoSeries.has_sindex`

[\[source\]](#)

Check the existence of the spatial index without generating it.

Use the `.sindex` attribute on a GeoDataFrame or GeoSeries to generate a spatial index if it does not yet exist, which may take considerable time based on the underlying index implementation.

Note that the underlying spatial index may not be fully initialized until the first use.

## Returns:

`bool`

*True* if the spatial index has been generated or *False* if not.

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d)
>>> gdf.has_sindex
False
>>> index = gdf.sindex
>>> gdf.has_sindex
True
```

# geopandas.GeoSeries.cx

## property GeoSeries.cx

[\[source\]](#)

Coordinate based indexer to select by intersection with bounding box.

Format of input should be `.cx[xmin:xmax, ymin:ymax]`. Any of `xmin`, `xmax`, `ymin`, and `ymax` can be provided, but input must include a comma separating x and y slices. That is, `.cx[:, :]` will return the full series/frame, but `.cx[:]` is not implemented.

## Examples

```
>>> from shapely.geometry import LineString, Point
>>> s = geopandas.GeoSeries(
... [Point(0, 0), Point(1, 2), Point(3, 3), LineString([(0, 0), (3, 3)])]
...)
>>> s
0 POINT (0 0)
1 POINT (1 2)
2 POINT (3 3)
3 LINESTRING (0 0, 3 3)
dtype: geometry
```

```
>>> s.cx[0:1, 0:1]
0 POINT (0 0)
3 LINESTRING (0 0, 3 3)
dtype: geometry
```

```
>>> s.cx[:, 1:]
1 POINT (1 2)
2 POINT (3 3)
3 LINESTRING (0 0, 3 3)
dtype: geometry
```

# geopandas.GeoSeries.\_\_geo\_interface\_\_

`property GeoSeries.__geo_interface__`

[\[source\]](#)

Returns a `GeoSeries` as a python feature collection.

Implements the `geo_interface`. The returned python data structure represents 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).

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)])
>>> s.__geo_interface__
{'type': 'FeatureCollection', 'features': [{"id": "0", "type": "Feature", "properties":
```

# GeoDataFrame

A `GeoDataFrame` is a tabular data structure that contains at least one `GeoSeries` column storing geometry.

## Constructor

`GeoDataFrame` ([data, geometry, crs])

A GeoDataFrame object is a pandas.DataFrame that has one or more columns containing geometry.

## Serialization / IO / conversion

`GeoDataFrame.from\_file` (filename, \*\*kwargs)

Alternate constructor to create a `GeoDataFrame` from a file.

`GeoDataFrame.from\_features` (features[, crs, ...])

Alternate constructor to create GeoDataFrame from an iterable of features or a feature collection.

`GeoDataFrame.from\_postgis` (sql, con[, ...])

Alternate constructor to create a `GeoDataFrame` from a sql query containing a geometry column in WKB representation.

`GeoDataFrame.from\_arrow` (table[, geometry])

Construct a GeoDataFrame from a Arrow table object based on GeoArrow extension types.

`GeoDataFrame.to\_file` (filename[, driver, ...])

Write the `GeoDataFrame` to a file.

`GeoDataFrame.to\_json` ([na, show\_bbox, ...])

Returns a GeoJSON representation of the `GeoDataFrame` as a string.

`GeoDataFrame.to\_geo\_dict` ([na, show\_bbox, ...])

Returns a python feature collection representation of the GeoDataFrame as a dictionary with a list of features based on the `\_\_geo\_interface\_\_` GeoJSON-like specification.

`GeoDataFrame.to\_parquet` (path[, index, ...])

Write a GeoDataFrame to the Parquet format.

`GeoDataFrame.to\_arrow` (\*[, index, ...])

Encode a GeoDataFrame to GeoArrow format.

<code>GeoDataFrame.to_feather</code> (path[, index, ...])	Write a GeoDataFrame to the Feather format.
<code>GeoDataFrame.to_postgis</code> (name, con[, schema, ...])	Upload GeoDataFrame into PostGIS database.
<code>GeoDataFrame.to_wkb</code> ([hex])	Encode all geometry columns in the GeoDataFrame to WKB.
<code>GeoDataFrame.to_wkt</code> (**kwargs)	Encode all geometry columns in the GeoDataFrame to WKT.

## Projection handling

<code>GeoDataFrame.crs</code>	The Coordinate Reference System (CRS) represented as a <code>pyproj.CRS</code> object.
<code>GeoDataFrame.set_crs</code> ([crs, epsg, inplace, ...])	Set the Coordinate Reference System (CRS) of the <code>GeoDataFrame</code> .
<code>GeoDataFrame.to_crs</code> ([crs, epsg, inplace])	Transform geometries to a new coordinate reference system.
<code>GeoDataFrame.estimate_utm_crs</code> ([datum_name])	Returns the estimated UTM CRS based on the bounds of the dataset.

## Active geometry handling

<code>GeoDataFrame.rename_geometry</code> (col[, inplace])	Renames the GeoDataFrame geometry column to the specified name.
<code>GeoDataFrame.set_geometry</code> (col[, drop, ...])	Set the GeoDataFrame geometry using either an existing column or the specified input.
<code>GeoDataFrame.active_geometry_name</code>	Return the name of the active geometry column

## Aggregating and exploding

<code>GeoDataFrame.dissolve</code> ([by, aggfunc, ...])	Dissolve geometries within <code>groupby</code> into single observation.
---------------------------------------------------------	--------------------------------------------------------------------------

[`GeoDataFrame.explode`](#)([column, ignore\_index, ...])

Explode multi-part geometries into multiple single geometries.

## Spatial joins

[`GeoDataFrame.sjoin`](#)(df, \*args, \*\*kwargs)

Spatial join of two GeoDataFrames.

[`GeoDataFrame.sjoin\_nearest`](#)(right[, how, ...])

Spatial join of two GeoDataFrames based on the distance between their geometries.

## Overlay operations

[`GeoDataFrame.clip`](#)(mask[, keep\_geom\_type, sort])

Clip points, lines, or polygon geometries to the mask extent.

[`GeoDataFrame.overlay`](#)(right[, how, ...])

Perform spatial overlay between GeoDataFrames.

## Plotting

[`GeoDataFrame.explore`](#)(\*args, \*\*kwargs)

Interactive map based on GeoPandas and folium/leaflet.js

[`GeoDataFrame.plot`](#)

alias of [`GeoplotAccessor`](#)

## Spatial index

[`GeoDataFrame.sindex`](#)

Generate the spatial index

[`GeoDataFrame.has\_sindex`](#)

Check the existence of the spatial index without generating it.

## Indexing

# Interface

## [`GeoDataFrame.\_\_geo\_interface\_\_`](#)

Returns a [GeoDataFrame](#) as a python feature collection.

## [`GeoDataFrame.iterfeatures`](#) ([na, show\_bbox, ...])

Returns an iterator that yields feature dictionaries that comply with `__geo_interface__`

All pandas [DataFrame](#) methods are also available, although they may not operate in a meaningful way on

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoDataFrame

```
class geopandas.GeoDataFrame(data=None, *args, geometry=None, crs=None, **kwargs)
```

[\[source\]](#)

A GeoDataFrame object is a pandas.DataFrame that has one or more columns containing geometry. In addition to the standard DataFrame constructor arguments, GeoDataFrame also accepts the following keyword arguments:

## Parameters:

### crs : value (optional)

Coordinate Reference System of the geometry objects. Can be anything accepted by [pyproj.CRS.from\\_user\\_input\(\)](#), such as an authority string (eg “EPSG:4326”) or a WKT string.

### geometry : str or array-like (optional)

Value to use as the active geometry column. If str, treated as column name to use. If array-like, it will be added as new column named ‘geometry’ on the GeoDataFrame and set as the active geometry column.

Note that if `geometry` is a (Geo)Series with a name, the name will not be used, a column named “geometry” will still be added. To preserve the name, you can use [rename\\_geometry\(\)](#) to update the geometry column name.

## See also

### [GeoSeries](#)

Series object designed to store shapely geometry objects

## Examples

Constructing GeoDataFrame from a dictionary.

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

Notice that the inferred dtype of ‘geometry’ columns is geometry.

```
>>> gdf.dtypes
col1 object
geometry geometry
dtype: object
```

Constructing GeoDataFrame from a pandas DataFrame with a column of WKT geometries:

```
>>> import pandas as pd
>>> d = {'col1': ['name1', 'name2'], 'wkt': ['POINT (1 2)', 'POINT (2 1)']}
>>> df = pd.DataFrame(d)
>>> gs = geopandas.GeoSeries.from_wkt(df['wkt'])
>>> gdf = geopandas.GeoDataFrame(df, geometry=gs, crs="EPSG:4326")
>>> gdf
 col1 wkt geometry
0 name1 POINT (1 2) POINT (1 2)
1 name2 POINT (2 1) POINT (2 1)
```

>>>

`__init__(data=None, *args, geometry=None, crs=None, **kwargs)`

[\[source\]](#)

## Methods

<code><u>__init__</u>([<i>data, geometry, crs</i>])</code>	
<code><u>abs</u>()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code><u>add</u>(<i>other[, axis, level, fill_value</i>])</code>	Get Addition of dataframe and other, element-wise (binary operator <i>add</i> ).
<code><u>add_prefix</u>(<i>prefix[, axis</i>)</code>	Prefix labels with string <i>prefix</i> .
<code><u>add_suffix</u>(<i>suffix[, axis</i>)</code>	Suffix labels with string <i>suffix</i> .
<code><u>affine_transform</u>(<i>matrix</i>)</code>	Return a <code>GeoSeries</code> with translated geometries.
<code><u>agg</u>([<i>func, axis</i>])</code>	Aggregate using one or more operations over the specified axis.
<code><u>aggregate</u>([<i>func, axis</i>])</code>	Aggregate using one or more operations over the specified axis.
<code><u>align</u>(<i>other[, join, axis, level, copy, ...]</i>)</code>	Align two objects on their axes with the specified join method.
<code><u>all</u>([<i>axis, bool_only, skipna</i>])</code>	Return whether all elements are True, potentially over an axis.

<code>any</code> (*[, axis, bool_only, skipna])	Return whether any element is True, potentially over an axis.
<code>apply</code> (func[, axis, raw, result_type, args])	Two-dimensional, size-mutable, potentially heterogeneous tabular data.
<code>applymap</code> (func[, na_action])	Apply a function to a Dataframe elementwise.
<code>asfreq</code> (freq[, method, how, normalize, ...])	Convert time series to specified frequency.
<code>asof</code> (where[, subset])	Return the last row(s) without any NaNs before <i>where</i> .
<code>assign</code> (**kwargs)	Assign new columns to a DataFrame.
<code>astype</code> (dtype[, copy, errors])	Cast a pandas object to a specified dtype <code>dtype</code> .
<code>at_time</code> (time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).
<code>backfill</code> (*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
<code>between_time</code> (start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill</code> (*[, axis, inplace, limit, limit_area, ...])	Fill NA/NaN values by using the next valid observation to fill the gap.
<code>bool</code> ()	Return the bool of a single element Series or DataFrame.
<code>boxplot</code> ([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame columns.
<code>buffer</code> (distance[, resolution, cap_style, ...])	Returns a <code>GeoSeries</code> of geometries representing all points within a given <code>distance</code> of each geometric object.
<code>build_area</code> ([node])	Creates an areal geometry formed by the constituent linework.
<code>clip</code> (mask[, keep_geom_type, sort])	Clip points, lines, or polygon geometries to the mask extent.

<code>clip_by_rect</code> (xmin, ymin, xmax, ymax)	Returns a <a href="#">GeoSeries</a> of the portions of geometry within the given rectangle.
<code>combine</code> (other, func[, fill_value, overwrite])	Perform column-wise combine with another DataFrame.
<code>combine_first</code> (other)	Update null elements with value in the same location in <i>other</i> .
<code>compare</code> (other[, align_axis, keep_shape, ...])	Compare to another DataFrame and show the differences.
<code>concave_hull</code> ([ratio, allow_holes])	Returns a <a href="#">GeoSeries</a> of geometries representing the concave hull of each geometry.
<code>contains</code> (other[, align])	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that contains <i>other</i> .
<code>contains_properly</code> (other[, align])	Returns a <a href="#">Series</a> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is completely inside <code>other</code> , with no common boundary points.
<code>convert_dtypes</code> ([infer_objects, ...])	Convert columns to the best possible dtypes using dtypes supporting <code>pd.NA</code> .
<code>copy</code> ([deep])	Two-dimensional, size-mutable, potentially heterogeneous tabular data.
<code>corr</code> ([method, min_periods, numeric_only])	Compute pairwise correlation of columns, excluding NA/null values.
<code>corrwith</code> (other[, axis, drop, method, ...])	Compute pairwise correlation.
<code>count</code> ([axis, numeric_only])	Count non-NA cells for each column or row.
<code>count_coordinates</code> ()	Returns a <a href="#">Series</a> containing the count of the number of coordinate pairs in each geometry.
<code>count_geometries</code> ()	Returns a <a href="#">Series</a> containing the count of geometries in each multi-part geometry.
<code>count_interior_rings</code> ()	Returns a <a href="#">Series</a> containing the count of the number of interior rings in a polygonal

	geometry.
<code>cov</code> ([min_periods, ddof, numeric_only])	Compute pairwise covariance of columns, excluding NA/null values.
<code>covered_by</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is entirely covered by <i>other</i> .
<code>covers</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is entirely covering <i>other</i> .
<code>crosses</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that cross <i>other</i> .
<code>cummax</code> ([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin</code> ([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod</code> ([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
<code>cumsum</code> ([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
<code>delaunay_triangles</code> ([tolerance, only_edges])	Returns a <code>GeoSeries</code> consisting of objects representing the computed Delaunay triangulation between the vertices of an input geometry.
<code>describe</code> ([percentiles, include, exclude])	Generate descriptive statistics.
<code>diff</code> ([periods, axis])	First discrete difference of element.
<code>difference</code> (other[, align])	Returns a <code>GeoSeries</code> of the points in each aligned geometry that are not in <i>other</i> .
<code>disjoint</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry disjoint to <i>other</i> .
<code>dissolve</code> ([by, aggfunc, as_index, level, ...])	Dissolve geometries within <code>groupby</code> into single observation.

<code>distance</code> (other[, align])	Returns a <code>Series</code> containing the distance to aligned <code>other</code> .
<code>div</code> (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator <code>truediv</code> ).
<code>divide</code> (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator <code>truediv</code> ).
<code>dot</code> (other)	Compute the matrix multiplication between the DataFrame and other.
<code>drop</code> ([labels, axis, index, columns, level, ...])	Drop specified labels from rows or columns.
<code>drop_duplicates</code> ([subset, keep, inplace, ...])	Return DataFrame with duplicate rows removed.
<code>droplevel</code> (level[, axis])	Return Series/DataFrame with requested index / column level(s) removed.
<code>dropna</code> (*[, axis, how, thresh, subset, ...])	Remove missing values.
<code>duplicated</code> ([subset, keep])	Return boolean Series denoting duplicate rows.
<code>dwithin</code> (other, distance[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is within a set distance from <code>other</code> .
<code>eq</code> (other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator <code>eq</code> ).
<code>equals</code> (other)	Test whether two objects contain the same elements.
<code>estimate_utm_crs</code> ([datum_name])	Returns the estimated UTM CRS based on the bounds of the dataset.
<code>eval</code> (expr, *[, inplace])	Evaluate a string describing operations on DataFrame columns.
<code>ewm</code> ([com, span, halflife, alpha, ...])	Provide exponentially weighted (EW) calculations.
<code>expanding</code> ([min_periods, axis, method])	Provide expanding window calculations.

<code><u>explode</u></code> ([column, ignore_index, index_parts])	Explode multi-part geometries into multiple single geometries.
<code><u>explore</u></code> (*args, **kwargs)	Interactive map based on GeoPandas and folium/leaflet.js
<code><u>extract_unique_points</u></code> ()	Returns a <code>GeoSeries</code> of MultiPoints representing all distinct vertices of an input geometry.
<code><u>ffill</u></code> (*[, axis, inplace, limit, limit_area, ...])	Fill NA/NaN values by propagating the last valid observation to next valid.
<code><u>fillna</u></code> ([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
<code><u>filter</u></code> ([items, like, regex, axis])	Subset the dataframe rows or columns according to the specified index labels.
<code><u>first</u></code> (offset)	Select initial periods of time series data based on a date offset.
<code><u>first_valid_index</u></code> ()	Return index for first non-NA value or None, if no non-NA value is found.
<code><u>floordiv</u></code> (other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator <i>floordiv</i> ).
<code><u>force_2d</u></code> ()	Forces the dimensionality of a geometry to 2D.
<code><u>force_3d</u></code> ([z])	Forces the dimensionality of a geometry to 3D.
<code><u>frechet_distance</u></code> (other[, align, densify])	Returns a <code>Series</code> containing the Frechet distance to aligned <i>other</i> .
<code><u>from_arrow</u></code> (table[, geometry])	Construct a GeoDataFrame from a Arrow table object based on GeoArrow extension types.
<code><u>from_dict</u></code> (data[, geometry, crs])	Construct GeoDataFrame from dict of array-like or dicts by overriding DataFrame.from_dict method with geometry and crs

<code>from_features</code> (features[, crs, columns])	Alternate constructor to create GeoDataFrame from an iterable of features or a feature collection.
<code>from_file</code> (filename, **kwargs)	Alternate constructor to create a GeoDataFrame from a file.
<code>from_postgis</code> (sql, con[, geom_col, crs, ...])	Alternate constructor to create a GeoDataFrame from a sql query containing a geometry column in WKB representation.
<code>from_records</code> (data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
<code>ge</code> (other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator ge).
<code>geom_almost_equals</code> (other[, decimal, align])	Returns a Series of dtype('bool') with value True if each aligned geometry is approximately equal to other.
<code>geom_equals</code> (other[, align])	Returns a Series of dtype('bool') with value True for each aligned geometry equal to other.
<code>geom_equals_exact</code> (other, tolerance[, align])	Return True for all geometries that equal aligned other to a given tolerance, else False.
<code>get</code> (key[, default])	Get item from object for given key (ex: DataFrame column).
<code>get_coordinates</code> ([include_z, ignore_index, ...])	Gets coordinates from a GeoSeries as a DataFrame of floats.
<code>get_geometry</code> (index)	Returns the n-th geometry from a collection of geometries.
<code>get_precision</code> ()	Returns a Series of the precision of each geometry.
<code>groupby</code> ([by, axis, level, as_index, sort, ...])	Group DataFrame using a mapper or by a Series of columns.

<code>gt</code> (other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator <i>gt</i> ).
<code>hausdorff_distance</code> (other[, align, densify])	Returns a <code>Series</code> containing the Hausdorff distance to aligned <i>other</i> .
<code>head</code> ([n])	Return the first <i>n</i> rows.
<code>hilbert_distance</code> ([total_bounds, level])	Calculate the distance along a Hilbert curve.
<code>hist</code> ([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's columns.
<code>idxmax</code> ([axis, skipna, numeric_only])	Return index of first occurrence of maximum over requested axis.
<code>idxmin</code> ([axis, skipna, numeric_only])	Return index of first occurrence of minimum over requested axis.
<code>infer_objects</code> ([copy])	Attempt to infer better dtypes for object columns.
<code>info</code> ([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a DataFrame.
<code>insert</code> (loc, column, value[, allow_duplicates])	Insert column into DataFrame at specified location.
<code>interpolate</code> (distance[, normalized])	Return a point at the specified distance along each geometry
<code>intersection</code> (other[, align])	Returns a <code>GeoSeries</code> of the intersection of points in each aligned geometry with <i>other</i> .
<code>intersection_all</code> ()	Returns a geometry containing the intersection of all geometries in the <code>GeoSeries</code> .
<code>intersects</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that intersects <i>other</i> .
<code>is_valid_reason</code> ()	Returns a <code>Series</code> of strings with the reason for invalidity of each geometry.
<code>isetitem</code> (loc, value)	Set the given value in the column with position <i>loc</i> .

<code>isin</code> (values)	Whether each element in the DataFrame is contained in values.
<code>isna</code> ()	Detect missing values.
<code>isnull</code> ()	DataFrame.isnull is an alias for DataFrame.isna.
<code>items</code> ()	Iterate over (column name, Series) pairs.
<code>iterfeatures</code> ([na, show_bbox, drop_id])	Returns an iterator that yields feature dictionaries that comply with <code>__geo_interface__</code>
<code>iterrows</code> ()	Iterate over DataFrame rows as (index, Series) pairs.
<code>itertuples</code> ([index, name])	Iterate over DataFrame rows as namedtuples.
<code>join</code> (other[, on, how, lsuffix, rsuffix, ...])	Join columns of another DataFrame.
<code>keys</code> ()	Get the 'info axis' (see Indexing for more).
<code>kurt</code> ([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
<code>kurtosis</code> ([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
<code>last</code> (offset)	Select final periods of time series data based on a date offset.
<code>last_valid_index</code> ()	Return index for last non-NA value or None, if no non-NA value is found.
<code>le</code> (other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise (binary operator <code>le</code> ).
<code>line_merge</code> ([directed])	Returns (Multi)LineStrings formed by combining the lines in a MultiLineString.
<code>lt</code> (other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator <code>lt</code> ).
<code>make_valid</code> ()	Repairs invalid geometries.

<code>map</code> (func[, na_action])	Apply a function to a Dataframe elementwise.
<code>mask</code> (cond[, other, inplace, axis, level])	Replace values where the condition is True.
<code>max</code> ([axis, skipna, numeric_only])	Return the maximum of the values over the requested axis.
<code>mean</code> ([axis, skipna, numeric_only])	Return the mean of the values over the requested axis.
<code>median</code> ([axis, skipna, numeric_only])	Return the median of the values over the requested axis.
<code>melt</code> ([id_vars, value_vars, var_name, ...])	Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
<code>memory_usage</code> ([index, deep])	Return the memory usage of each column in bytes.
<code>merge</code> (right[, how, on, left_on, right_on, ...])	Merge DataFrame or named Series objects with a database-style join.
<code>min</code> ([axis, skipna, numeric_only])	Return the minimum of the values over the requested axis.
<code>minimum_bounding_circle</code> ()	Returns a <code>GeoSeries</code> of geometries representing the minimum bounding circle that encloses each geometry.
<code>minimum_bounding_radius</code> ()	Returns a <code>Series</code> of the radii of the minimum bounding circles that enclose each geometry.
<code>minimum_clearance</code> ()	Returns a <code>Series</code> containing the minimum clearance distance, which is the smallest distance by which a vertex of the geometry could be moved to produce an invalid geometry.
<code>minimum_rotated_rectangle</code> ()	Returns a <code>GeoSeries</code> of the general minimum bounding rectangle that contains the object.
<code>mod</code> (other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator <i>mod</i> ).

<code>mode</code> ([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
<code>mul</code> (other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator <i>mul</i> ).
<code>multiply</code> (other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator <i>mul</i> ).
<code>ne</code> (other[, axis, level])	Get Not equal to of dataframe and other, element-wise (binary operator <i>ne</i> ).
<code>nlargest</code> (n, columns[, keep])	Return the first <i>n</i> rows ordered by <i>columns</i> in descending order.
<code>normalize</code> ()	Returns a <code>GeoSeries</code> of normalized geometries to normal form (or canonical form).
<code>notna</code> ()	Detect existing (non-missing) values.
<code>notnull</code> ()	DataFrame.notnull is an alias for DataFrame.notna.
<code>nsmallest</code> (n, columns[, keep])	Return the first <i>n</i> rows ordered by <i>columns</i> in ascending order.
<code>nunique</code> ([axis, dropna])	Count number of distinct elements in specified axis.
<code>offset_curve</code> (distance[, quad_segs, ...])	Returns a <code>LineString</code> or <code>MultiLineString</code> geometry at a distance from the object on its right or its left side.
<code>overlaps</code> (other[, align])	Returns True for all aligned geometries that overlap <i>other</i> , else False.
<code>overlay</code> (right[, how, keep_geom_type, make_valid])	Perform spatial overlay between GeoDataFrames.
<code>pad</code> (*[, axis, inplace, limit, downcast])	Fill NA/Nan values by propagating the last valid observation to next valid.
<code>pct_change</code> ([periods, fill_method, limit, freq])	Fractional change between the current and a prior element.

<code>pipe</code> (func, *args, **kwargs)	Apply chainable functions that expect Series or DataFrames.
<code>pivot</code> (*[, columns[, index, values]])	Return reshaped DataFrame organized by given index / column values.
<code>pivot_table</code> ([values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
<code>polygonize</code> ([node, full])	Creates polygons formed from the linework of a GeoSeries.
<code>pop</code> (item)	Return item and drop from frame.
<code>pow</code> (other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator <i>pow</i> ).
<code>prod</code> ([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
<code>product</code> ([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
<code>project</code> (other[, normalized, align])	Return the distance along each geometry nearest to <i>other</i>
<code>quantile</code> ([q, axis, numeric_only, ...])	Return values at the given quantile over requested axis.
<code>query</code> (expr, *[, inplace])	Query the columns of a DataFrame with a boolean expression.
<code>radd</code> (other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator <i>radd</i> ).
<code>rank</code> ([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
<code>rdiv</code> (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).
<code>reindex</code> ([labels, index, columns, axis, ...])	Conform DataFrame to new index with optional filling logic.
<code>reindex_like</code> (other[, method, copy, limit, ...])	Return an object with matching indices as other object.

<code>relate</code> (other[, align])	Returns the DE-9IM intersection matrices for the geometries
<code>relate_pattern</code> (other, pattern[, align])	Returns True if the DE-9IM string code for the relationship between the geometries satisfies the pattern, else False.
<code>remove_repeated_points</code> ([tolerance])	Returns a <code>GeoSeries</code> containing a copy of the input geometry with repeated points removed.
<code>rename</code> ([mapper, index, columns, axis, copy, ...])	Rename columns or index labels.
<code>rename_axis</code> ([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
<code>rename_geometry</code> (col[, inplace])	Renames the GeoDataFrame geometry column to the specified name.
<code>reorder_levels</code> (order[, axis])	Rearrange index levels using input order.
<code>replace</code> ([to_replace, value, inplace, limit, ...])	Replace values given in <i>to_replace</i> with <i>value</i> .
<code>representative_point</code> ()	Returns a <code>GeoSeries</code> of (cheaply computed) points that are guaranteed to be within each geometry.
<code>resample</code> (rule[, axis, closed, label, ...])	Resample time-series data.
<code>reset_index</code> ([level, drop, inplace, ...])	Reset the index, or a level of it.
<code>reverse</code> ()	Returns a <code>GeoSeries</code> with the order of coordinates reversed.
<code>rfloordiv</code> (other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator <i>rfloordiv</i> ).
<code>rmod</code> (other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator <i>rmod</i> ).
<code>rmul</code> (other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator <i>rmul</i> ).
<code>rolling</code> (window[, min_periods, center, ...])	Provide rolling window calculations.

<code>rotate</code> (angle[, origin, use_radians])	Returns a <a href="#">GeoSeries</a> with rotated geometries.
<code>round</code> ([decimals])	Round a DataFrame to a variable number of decimal places.
<code>rpow</code> (other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator <i>rpow</i> ).
<code>rsub</code> (other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator <i>rsub</i> ).
<code>rtruediv</code> (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).
<code>sample</code> ([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
<code>sample_points</code> (size[, method, seed, rng])	Sample points from each geometry.
<code>scale</code> ([xfact, yfact, zfact, origin])	Returns a <a href="#">GeoSeries</a> with scaled geometries.
<code>segmentize</code> (max_segment_length)	Returns a <a href="#">GeoSeries</a> with vertices added to line segments based on maximum segment length.
<code>select_dtypes</code> ([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
<code>sem</code> ([axis, skipna, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
<code>set_axis</code> (labels, *[, axis, copy])	Assign desired index to given axis.
<code>set_crs</code> ([crs, epsg, inplace, allow_override])	Set the Coordinate Reference System (CRS) of the <a href="#">GeoDataFrame</a> .
<code>set_flags</code> (*[, copy, allows_duplicate_labels])	Return a new object with updated flags.
<code>set_geometry</code> (col[, drop, inplace, crs])	Set the GeoDataFrame geometry using either an existing column or the specified input.
<code>set_index</code> (keys, *[, drop, append, inplace, ...])	Set the DataFrame index using existing columns.

<code>set_precision</code> (grid_size[, mode])	Returns a <code>GeoSeries</code> with the precision set to a precision grid size.
<code>shared_paths</code> (other[, align])	Returns the shared paths between two geometries.
<code>shift</code> ([periods, freq, axis, fill_value, suffix])	Shift index by desired number of periods with an optional time <i>freq</i> .
<code>shortest_line</code> (other[, align])	Returns the shortest two-point line between two geometries.
<code>simplify</code> (tolerance[, preserve_topology])	Returns a <code>GeoSeries</code> containing a simplified representation of each geometry.
<code>sjoin</code> (df, *args, **kwargs)	Spatial join of two GeoDataFrames.
<code>sjoin_nearest</code> (right[, how, max_distance, ...])	Spatial join of two GeoDataFrames based on the distance between their geometries.
<code>skew</code> ([xs, ys, origin, use_radians])	Returns a <code>GeoSeries</code> with skewed geometries.
<code>snap</code> (other, tolerance[, align])	Snaps an input geometry to reference geometry's vertices.
<code>sort_index</code> (*[, axis, level, ascending, ...])	Sort object by labels (along an axis).
<code>sort_values</code> (by, *[, axis, ascending, ...])	Sort by the values along either axis.
<code>squeeze</code> ([axis])	Squeeze 1 dimensional axis objects into scalars.
<code>stack</code> ([level, dropna, sort, future_stack])	Stack the prescribed level(s) from columns to index.
<code>std</code> ([axis, skipna, ddof, numeric_only])	Return sample standard deviation over requested axis.
<code>sub</code> (other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator <i>sub</i> ).
<code>subtract</code> (other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator <i>sub</i> ).
<code>sum</code> ([axis, skipna, numeric_only, min_count])	Return the sum of the values over the requested axis.

<code>swapaxes</code> (axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
<code>swaplevel</code> ([i, j, axis])	Swap levels i and j in a <code>MultiIndex</code> .
<code>symmetric_difference</code> (other[, align])	Returns a <code>GeoSeries</code> of the symmetric difference of points in each aligned geometry with <i>other</i> .
<code>tail</code> ([n])	Return the last n rows.
<code>take</code> (indices[, axis])	Return the elements in the given <i>positional</i> indices along an axis.
<code>to_arrow</code> (*[, index, geometry_encoding, ...])	Encode a GeoDataFrame to GeoArrow format.
<code>to_clipboard</code> (*[, excel, sep])	Copy object to the system clipboard.
<code>to_crs</code> ([crs, epsg, inplace])	Transform geometries to a new coordinate reference system.
<code>to_csv</code> ([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
<code>to_dict</code> ([orient, into, index])	Convert the DataFrame to a dictionary.
<code>to_excel</code> (excel_writer, *[, sheet_name, ...])	Write object to an Excel sheet.
<code>to_feather</code> (path[, index, compression, ...])	Write a GeoDataFrame to the Feather format.
<code>to_file</code> (filename[, driver, schema, index])	Write the <code>GeoDataFrame</code> to a file.
<code>to_gbq</code> (destination_table, *[, project_id, ...])	Write a DataFrame to a Google BigQuery table.
<code>to_geo_dict</code> ([na, show_bbox, drop_id])	Returns a python feature collection representation of the GeoDataFrame as a dictionary with a list of features based on the <code>__geo_interface__</code> GeoJSON-like specification.
<code>to_hdf</code> (path_or_buf, *, key[, mode, ...])	Write the contained data to an HDF5 file using HDFStore.
<code>to_html</code> ([buf, columns, col_space, header, ...])	Render a DataFrame as an HTML table.

<code><a href="#">to_json</a></code> ([na, show_bbox, drop_id, to_wgs84])	Returns a GeoJSON representation of the <code><a href="#">GeoDataFrame</a></code> as a string.
<code><a href="#">to_latex</a></code> ([buf, columns, header, index, ...])	Render object to a LaTeX tabular, longtable, or nested table.
<code><a href="#">to_markdown</a></code> ([buf, mode, index, storage_options])	Print DataFrame in Markdown-friendly format.
<code><a href="#">to_numpy</a></code> ([dtype, copy, na_value])	Convert the DataFrame to a NumPy array.
<code><a href="#">to_orc</a></code> ([path, engine, index, engine_kwargs])	Write a DataFrame to the ORC format.
<code><a href="#">to_parquet</a></code> (path[, index, compression, ...])	Write a GeoDataFrame to the Parquet format.
<code><a href="#">to_period</a></code> ([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex.
<code><a href="#">to_pickle</a></code> (path, *[, compression, protocol, ...])	Pickle (serialize) object to file.
<code><a href="#">to_postgis</a></code> (name, con[, schema, if_exists, ...])	Upload GeoDataFrame into PostGIS database.
<code><a href="#">to_records</a></code> ([index, column_dtypes, index_dtypes])	Convert DataFrame to a NumPy record array.
<code><a href="#">to_sql</a></code> (name, con, *[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
<code><a href="#">to_stata</a></code> (path, *[, convert_dates, ...])	Export DataFrame object to Stata dta format.
<code><a href="#">to_string</a></code> ([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
<code><a href="#">to_timestamp</a></code> ([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at <i>beginning</i> of period.
<code><a href="#">to_wkb</a></code> ([hex])	Encode all geometry columns in the GeoDataFrame to WKB.
<code><a href="#">to_wkt</a></code> (**kwargs)	Encode all geometry columns in the GeoDataFrame to WKT.
<code><a href="#">to_xarray</a></code> ()	Return an xarray object from the pandas object.

<code>to_xml</code> ([path_or_buffer, index, root_name, ...])	Render a DataFrame to an XML document.
<code>touches</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that touches <code>other</code> .
<code>transform</code> (transformation[, include_z])	Returns a <code>GeoSeries</code> with the transformation function applied to the geometry coordinates.
<code>translate</code> ([xoff, yoff, zoff])	Returns a <code>GeoSeries</code> with translated geometries.
<code>transpose</code> (*args[, copy])	Transpose index and columns.
<code>truediv</code> (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>truncate</code> ([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
<code>tz_convert</code> (tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
<code>tz_localize</code> (tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
<code>union</code> (other[, align])	Returns a <code>GeoSeries</code> of the union of points in each aligned geometry with <code>other</code> .
<code>union_all</code> ([method])	Returns a geometry containing the union of all geometries in the <code>GeoSeries</code> .
<code>unstack</code> ([level, fill_value, sort])	Pivot a level of the (necessarily hierarchical) index labels.
<code>update</code> (other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.
<code>value_counts</code> ([subset, normalize, sort, ...])	Return a Series containing the frequency of each distinct row in the Dataframe.
<code>var</code> ([axis, skipna, ddof, numeric_only])	Return unbiased variance over requested axis.
<code>voronoi_polygons</code> ([tolerance, extend_to, ...])	Returns a <code>GeoSeries</code> consisting of objects representing the computed Voronoi diagram around the vertices of an input geometry.

<code>where</code> (cond[, other, inplace, axis, level])	Replace values where the condition is False.
<code>within</code> (other[, align])	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for each aligned geometry that is within <code>other</code> .
<code>xs</code> (key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.

## Attributes

<code>T</code>	The transpose of the DataFrame.
<code>active_geometry_name</code>	Return the name of the active geometry column
<code>area</code>	Returns a <code>Series</code> containing the area of each geometry in the <code>GeoSeries</code> expressed in the units of the CRS.
<code>at</code>	Access a single value for a row/column label pair.
<code>attrs</code>	Dictionary of global attributes of this dataset.
<code>axes</code>	Return a list representing the axes of the DataFrame.
<code>boundary</code>	Returns a <code>GeoSeries</code> of lower dimensional objects representing each geometry's set-theoretic <i>boundary</i> .
<code>bounds</code>	Returns a <code>DataFrame</code> with columns <code>minx</code> , <code>miny</code> , <code>maxx</code> , <code>maxy</code> values containing the bounds for each geometry.
<code>centroid</code>	Returns a <code>GeoSeries</code> of points representing the centroid of each geometry.
<code>columns</code>	The column labels of the DataFrame.
<code>convex_hull</code>	Returns a <code>GeoSeries</code> of geometries representing the convex hull of each geometry.
<code>crs</code>	The Coordinate Reference System (CRS) represented as a <code>pyproj.CRS</code> object.
<code>cx</code>	Coordinate based indexer to select by intersection with bounding box.
<code>dtypes</code>	Return the dtypes in the DataFrame.
<code>empty</code>	Indicator whether Series/DataFrame is empty.

<code>envelope</code>	Returns a <code>GeoSeries</code> of geometries representing the envelope of each geometry.
<code>exterior</code>	Returns a <code>GeoSeries</code> of LinearRings representing the outer boundary of each polygon in the GeoSeries.
<code>flags</code>	Get the properties associated with this pandas object.
<code>geom_type</code>	Returns a <code>Series</code> of strings specifying the <i>Geometry Type</i> of each object.
<code>geometry</code>	Geometry data for GeoDataFrame
<code>has_sindex</code>	Check the existence of the spatial index without generating it.
<code>has_z</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for features that have a z-component.
<code>iat</code>	Access a single value for a row/column pair by integer position.
<code>iloc</code>	Purely integer-location based indexing for selection by position.
<code>index</code>	The index (row labels) of the DataFrame.
<code>interiors</code>	Returns a <code>Series</code> of List representing the inner rings of each polygon in the GeoSeries.
<code>is_ccw</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> if a LineString or LinearRing is counterclockwise.
<code>is_closed</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> if a LineString's or LinearRing's first and last points are equal.
<code>is_empty</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for empty geometries.
<code>is_ring</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for features that are closed.
<code>is_simple</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for geometries that do not cross themselves.
<code>is_valid</code>	Returns a <code>Series</code> of <code>dtype('bool')</code> with value <code>True</code> for geometries that are valid.

# geopandas.GeoDataFrame.from\_file

*class***method** `GeoDataFrame.from_file(filename, **kwargs)`

[\[source\]](#)

Alternate constructor to create a `GeoDataFrame` from a file.

It is recommended to use `geopandas.read_file()` instead.

Can load a `GeoDataFrame` from a file in any format recognized by `pyogrio`. See <http://pyogrio.readthedocs.io/> for details.

## Parameters:

`filename : str`

File path or file handle to read from. Depending on which kwargs are included, the content of filename may vary. See `pyogrio.read_dataframe()` for usage details.

`kwargs : key-word arguments`

These arguments are passed to `pyogrio.read_dataframe()`, and can be used to access multi-layer data, data stored within archives (zip files), etc.

## See also

`read_file`

read file to GeoDataFrame

`GeoDataFrame.to_file`

write GeoDataFrame to file

## Examples

```
>>> import geodatasets
>>> path = geodatasets.get_path('nybb')
>>> gdf = geopandas.GeoDataFrame.from_file(path)
>>> gdf
 BoroCode BoroName Shape_Leng Shape_Area
0 5 Staten Island 330470.010332 1.623820e+09 MULTIPOLYGON (((970217.022 1
1 4 Queens 896344.047763 3.045213e+09 MULTIPOLYGON (((1029606.077
2 3 Brooklyn 741080.523166 1.937479e+09 MULTIPOLYGON (((1021176.479
3 1 Manhattan 359299.096471 6.364715e+08 MULTIPOLYGON (((981219.056 1
4 2 Bronx 464392.991824 1.186925e+09 MULTIPOLYGON (((1012821.806
```

The recommended method of reading files is `geopandas.read_file()`:

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoDataFrame.from\_features

*classmethod* `GeoDataFrame.from_features(features, crs=None, columns=None)`

Alternate constructor to create GeoDataFrame from an iterable of features or a feature collection.

[\[source\]](#)

## Parameters:

### `features`

- Iterable of features, where each element must be a feature dictionary or implement the `__geo_interface__`.
- Feature collection, where the ‘features’ key contains an iterable of features.
- Object holding a feature collection that implements the `__geo_interface__`.

### `crs : str or dict (optional)`

Coordinate reference system to set on the resulting frame.

### `columns : list of column names, optional`

Optionally specify the column names to include in the output frame. This does not overwrite the property names of the input, but can ensure a consistent output format.

## Returns:

### `GeoDataFrame`

## Notes

For more information about the `__geo_interface__`, see <https://gist.github.com/sgillies/2217756>

## Examples

```
>>> feature_coll = {
... "type": "FeatureCollection",
... "features": [
... {
... "id": "0",
... "type": "Feature",
... "properties": {"col1": "name1"},
... "geometry": {"type": "Point", "coordinates": (1.0, 2.0)},
... "bbox": (1.0, 2.0, 1.0, 2.0),
... }
>>>
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoDataFrame.from\_arrow

*classmethod* `GeoDataFrame.from_arrow(table, geometry=None)`

[\[source\]](#)

Construct a GeoDataFrame from a Arrow table object based on GeoArrow extension types.

See <https://geoarrow.org/> for details on the GeoArrow specification.

This function accepts any tabular Arrow object implementing the [Arrow PyCapsule Protocol](#) (i.e. having an `__arrow_c_array__` or `__arrow_c_stream__` method).

 **Added in version 1.0.**

## Parameters:

**table : `pyarrow.Table` or Arrow-compatible table**

Any tabular object implementing the Arrow PyCapsule Protocol (i.e. has an `__arrow_c_array__` or `__arrow_c_stream__` method). This table should have at least one column with a geoarrow geometry type.

**geometry : str, default None**

The name of the geometry column to set as the active geometry column. If None, the first geometry column found will be used.

## Returns:

`GeoDataFrame`

# geopandas.GeoDataFrame.to\_file

`GeoDataFrame.to_file(filename, driver=None, schema=None, index=None, **kwargs)`

[\[source\]](#)

Write the `GeoDataFrame` to a file.

By default, an ESRI shapefile is written, but any OGR data source supported by Pyogrio or Fiona can be written. A dictionary of supported OGR providers is available via:

```
>>> import pyogrio
>>> pyogrio.list_drivers()
```

## Parameters:

**filename : string**

File path or file handle to write to. The path may specify a GDAL VSI scheme.

**driver : string, default None**

The OGR format driver used to write the vector file. If not specified, it attempts to infer it from the file extension. If no extension is specified, it saves ESRI Shapefile to a folder.

**schema : dict, default None**

If specified, the schema dictionary is passed to Fiona to better control how the file is written.

If None, GeoPandas will determine the schema based on each column's dtype. Not supported for the "pyogrio" engine.

**index : bool, default None**

If True, write index into one or more columns (for MultiIndex). Default None writes the index into one or more columns only if the index is named, is a MultiIndex, or has a non-integer data type. If False, no index is written.

 **Added in version 0.7:** Previously the index was not written.

**mode : string, default 'w'**

The write mode, 'w' to overwrite the existing file and 'a' to append. Not all drivers support appending. The drivers that support appending are listed in `fiona.supported_drivers` or

 [Toblerity/Fiona](#)

**crs : pyproj.CRS, default None**

If specified, the CRS is passed to Fiona to better control how the file is written. If None, GeoPandas will determine the crs based on crs df attribute. The value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg "EPSG:4326") or a WKT string. The keyword is not supported for the "pyogrio" engine.

## **engine : str, “pyogrio” or “fiona”**

The underlying library that is used to write the file. Currently, the supported options are “pyogrio” and “fiona”. Defaults to “pyogrio” if installed, otherwise tries “fiona”.

## **metadata : dict[str, str], default None**

Optional metadata to be stored in the file. Keys and values must be strings. Supported only for “GPKG” driver.

## **\*\*kwargs**

Keyword args to be passed to the engine, and can be used to write to multi-layer data, store data within archives (zip files), etc. In case of the “pyogrio” engine, the keyword arguments are passed to `pyogrio.write_dataframe`. In case of the “fiona” engine, the keyword arguments are passed to `fiona.open``. For more information on possible keywords, type: `import pyogrio; help(pyogrio.write_dataframe)`.

## See also

`GeoSeries.to\_file`

`GeoDataFrame.to\_postgis`

write GeoDataFrame to PostGIS database

`GeoDataFrame.to\_parquet`

write GeoDataFrame to parquet

`GeoDataFrame.to\_feather`

write GeoDataFrame to feather

## Notes

The format drivers will attempt to detect the encoding of your data, but may fail. In this case, the proper encoding can be specified explicitly by using the encoding keyword parameter, e.g.

`encoding='utf-8'`.

## Examples

```
>>> gdf.to_file('dataframe.shp')
```

>>>

```
>>> gdf.to_file('dataframe.gpkg', driver='GPKG', layer='name')
```

>>>

```
>>> gdf.to_file('dataframe.geojson', driver='GeoJSON')
```

>>>

With selected drivers you can also append to a file with `mode="a"`:

```
>>> gdf.to_file('dataframe.shp', mode="a")
```

>>>

Using the engine-specific keyword arguments it is possible to e.g. create a spatialite file with a custom layer name:

```
>>> gdf.to_file(
... 'dataframe.sqlite', driver='SQLLite', spatialite=True, layer='test'
...)
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.to\_json

`GeoDataFrame.to_json(na='null', show_bbox=False, drop_id=False, to_wgs84=False, **kwargs)`

[\[source\]](#)

Returns a GeoJSON representation of the `GeoDataFrame` as a string.

## Parameters:

`na : {'null', 'drop', 'keep'}, default 'null'`

Indicates how to output missing (NaN) values in the GeoDataFrame. See below.

`show_bbox : bool, optional, default: False`

Include bbox (bounds) in the geojson

`drop_id : bool, default: False`

Whether to retain the index of the GeoDataFrame as the id property in the generated GeoJSON. Default is False, but may want True if the index is just arbitrary row numbers.

`to_wgs84: bool, optional, default: False`

If the CRS is set on the active geometry column it is exported as WGS84 (EPSG:4326) to meet the [2016 GeoJSON specification](#). Set to True to force re-projection and set to False to ignore CRS. False by default.

## See also

[`GeoDataFrame.to\_file`](#)

write GeoDataFrame to file

## Notes

The remaining `kwargs` are passed to `json.dumps()`.

Missing (NaN) values in the GeoDataFrame can be represented as follows:

- `null`: output the missing entries as JSON null.
- `drop`: remove the property from the feature. This applies to each feature individually so that features may have different properties.
- `keep`: output the missing entries as NaN.

If the GeoDataFrame has a defined CRS, its definition will be included in the output unless it is equal to WGS84 (default GeoJSON CRS) or not possible to represent in the URN OGC format, or unless `to_wgs84=True` is specified.

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:3857")
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

```
>>> gdf.to_json()
'{"type": "FeatureCollection", "features": [{"id": "0", "type": "Feature", "properties": {}}, {"id": "1", "type": "Feature", "properties": {}}]}
```

Alternatively, you can write GeoJSON to file:

```
>>> gdf.to_file(path, driver="GeoJSON")
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.to\_geodict

`GeoDataFrame.to_geodict(na='null', show_bbox=False, drop_id=False)` [\[source\]](#)

Returns a python feature collection representation of the GeoDataFrame as a dictionary with a list of features based on the [\\_\\_geo\\_interface\\_\\_](#) GeoJSON-like specification.

## Parameters:

### `na : str, optional`

Options are {'null', 'drop', 'keep'}, default 'null'. Indicates how to output missing (NaN) values in the GeoDataFrame

- null: output the missing entries as JSON null
- drop: remove the property from the feature. This applies to each feature individually so that features may have different properties
- keep: output the missing entries as NaN

### `show_bbox : bool, optional`

Include bbox (bounds) in the geojson. Default False.

### `drop_id : bool, default: False`

Whether to retain the index of the GeoDataFrame as the id property in the generated dictionary. Default is False, but may want True if the index is just arbitrary row numbers.

## See also

### [GeoDataFrame.to\\_json](#)

return a GeoDataFrame as a GeoJSON string

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d)
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

```
>>> gdf.to_geodict()
{'type': 'FeatureCollection', 'features': [{"id": "0", "type": "Feature", "properties":
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoDataFrame.to\_arrow

```
GeoDataFrame.to_arrow(*, index=None, geometry_encoding='WKB',
interleaved=True, include_z=None)
```

[\[source\]](#)

Encode a GeoDataFrame to GeoArrow format.

See <https://geoarrow.org/> for details on the GeoArrow specification.

This function returns a generic Arrow data object implementing the [Arrow PyCapsule Protocol](#) (i.e. having an `__arrow_c_stream__` method). This object can then be consumed by your Arrow implementation of choice that supports this protocol.

**!** *Added in version 1.0.*

## Parameters:

**index : bool, default None**

If `True`, always include the dataframe's index(es) as columns in the file output. If `False`, the index(es) will not be written to the file. If `None`, the index(es) will be included as columns in the file output except `RangeIndex` which is stored as metadata only.

**geometry\_encoding : {'WKB', 'geoarrow'}, default 'WKB'**

The GeoArrow encoding to use for the data conversion.

**interleaved : bool, default True**

Only relevant for 'geoarrow' encoding. If True, the geometries' coordinates are interleaved in a single fixed size list array. If False, the coordinates are stored as separate arrays in a struct type.

**include\_z : bool, default None**

Only relevant for 'geoarrow' encoding (for WKB, the dimensionality of the individual geometries is preserved). If False, return 2D geometries. If True, include the third dimension in the output (if a geometry has no third dimension, the z-coordinates will be NaN). By default, will infer the dimensionality from the input geometries. Note that this inference can be unreliable with empty geometries (for a guaranteed result, it is recommended to specify the keyword).

## Returns:

**ArrowTable**

A generic Arrow table object with geometry columns encoded to GeoArrow.

## Examples

```
>>> from shapely.geometry import Point
>>> data = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(data)
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

```
>>> arrow_table = gdf.to_arrow()
>>> arrow_table
<geopandas.io._geoarrow.ArrowTable object at ...>
```

The returned data object needs to be consumed by a library implementing the Arrow PyCapsule Protocol. For example, wrapping the data as a pyarrow.Table (requires pyarrow >= 14.0):

```
>>> import pyarrow as pa
>>> table = pa.table(arrow_table)
>>> table
pyarrow.Table
col1: string
geometry: binary

col1: [["name1", "name2"]]
geometry: [[010100000000000000000000F03F000000000000000040, 010100000000000000000000400000]
```

# geopandas.GeoDataFrame.to\_feather

`GeoDataFrame.to_feather(path, index=None, compression=None, schema_version=None, **kwargs)`

[\[source\]](#)

Write a GeoDataFrame to the Feather format.

Any geometry columns present are serialized to WKB format in the file.

Requires ‘pyarrow’ >= 0.17.

 **Added in version 0.8.**

## Parameters:

`path : str, path object`

`index : bool, default None`

If `True`, always include the dataframe’s index(es) as columns in the file output. If `False`, the index(es) will not be written to the file. If `None`, the index(ex) will be included as columns in the file output except `RangeIndex` which is stored as metadata only.

`compression : {'zstd', 'lz4', 'uncompressed'}, optional`

Name of the compression to use. Use `"uncompressed"` for no compression. By default uses LZ4 if available, otherwise uncompressed.

`schema_version : {'0.1.0', '0.4.0', '1.0.0', None}`

GeoParquet specification version; if not provided will default to latest supported version.

`kwargs`

Additional keyword arguments passed to to [`pyarrow.feather.write\_feather\(\)`](#).

## See also

[`GeoDataFrame.to\_parquet`](#)

write GeoDataFrame to parquet

[`GeoDataFrame.to\_file`](#)

write GeoDataFrame to file

## Examples



# geopandas.GeoDataFrame.to\_postgis

`GeoDataFrame.to_postgis(name, con, schema=None, if_exists='fail', index=False, index_label=None, chunksize=None, dtype=None)`

[\[source\]](#)

Upload GeoDataFrame into PostGIS database.

This method requires SQLAlchemy and GeoAlchemy2, and a PostgreSQL Python driver (psycopg or psycopg2) to be installed.

It is also possible to use [to\\_file\(\)](#) to write to a database. Especially for file geodatabases like GeoPackage or SpatiaLite this can be easier.

## Parameters:

**name : str**

Name of the target table.

**con : sqlalchemy.engine.Connection or sqlalchemy.engine.Engine**

Active connection to the PostGIS database.

**if\_exists : {'fail', 'replace', 'append'}, default 'fail'**

How to behave if the table already exists:

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

**schema : string, optional**

Specify the schema. If None, use default schema: 'public'.

**index : bool, default False**

Write DataFrame index as a column. Uses `index_label` as the column name in the table.

**index\_label : string or sequence, default None**

Column label for index column(s). If None is given (default) and index is True, then the index names are used.

**chunksize : int, optional**

Rows will be written in batches of this size at a time. By default, all rows will be written at once.

**dtype : dict of column name to SQL type, default None**

Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types.

## See also

[`GeoDataFrame.to\_file`](#)

write GeoDataFrame to file

[`read\_postgis`](#)

read PostGIS database to GeoDataFrame

## Examples

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine("postgresql://myusername:mypassword@myhost:5432/mydatabase")
>>> gdf.to_postgis("my_table", engine)
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.to\_wkb

`GeoDataFrame.to_wkb(hex=False, **kwargs)`

[\[source\]](#)

Encode all geometry columns in the GeoDataFrame to WKB.

## Parameters:

`hex : bool`

If true, export the WKB as a hexadecimal string. The default is to return a binary bytes object.

`kwargs`

Additional keyword args will be passed to [`shapely.to\_wkb\(\)`](#).

## Returns:

`DataFrame`

geometry columns are encoded to WKB



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.GeoDataFrame.crs

*property* `GeoDataFrame.crs`

[\[source\]](#)

The Coordinate Reference System (CRS) represented as a `pyproj.CRS` object.

Returns None if the CRS is not set, and to set the value it :getter: Returns a `pyproj.CRS` or None.

When setting, the value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

## See also

[`GeoDataFrame.set\_crs`](#)

assign CRS

[`GeoDataFrame.to\_crs`](#)

re-project to another CRS

## Examples

```
>>> gdf.crs
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

# geopandas.GeoDataFrame.set\_crs

`GeoDataFrame.set_crs(crs=None, epsg=None, inplace=False, allow_override=False)`

[\[source\]](#)

Set the Coordinate Reference System (CRS) of the `GeoDataFrame`.

If there are multiple geometry columns within the GeoDataFrame, only the CRS of the active geometry column is set.

Pass `None` to remove CRS from the active geometry column.

## Parameters:

**crs : *pyproj.CRS | None, optional***

The value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

**epsg : *int, optional***

EPSG code specifying the projection.

**inplace : *bool, default False***

If True, the CRS of the GeoDataFrame will be changed in place (while still returning the result) instead of making a copy of the GeoDataFrame.

**allow\_override : *bool, default False***

If the the GeoDataFrame already has a CRS, allow to replace the existing CRS, even when both are not equal.

## See also

[`GeoDataFrame.to\_crs`](#)

re-project to another CRS

## Notes

The underlying geometries are not transformed to this CRS. To transform the geometries to a new CRS, use the `to_crs` method.

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d)
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

Setting CRS to a GeoDataFrame without one:

```
>>> gdf.crs is None
True
```

```
>>> gdf = gdf.set_crs('epsg:3857')
>>> gdf.crs
<Projected CRS: EPSG:3857>
Name: WGS 84 / Pseudo-Mercator
Axis Info [cartesian]:
- X[east]: Easting (metre)
- Y[north]: Northing (metre)
Area of Use:
- name: World - 85°S to 85°N
- bounds: (-180.0, -85.06, 180.0, 85.06)
Coordinate Operation:
- name: Popular Visualisation Pseudo-Mercator
- method: Popular Visualisation Pseudo Mercator
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Overriding existing CRS:

```
>>> gdf = gdf.set_crs(4326, allow_override=True)
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.to\_crs

`GeoDataFrame.to_crs(crs=None, epsg=None, inplace=False)`

[\[source\]](#)

Transform geometries to a new coordinate reference system.

Transform all geometries in an active geometry column to a different coordinate reference system. The `crs` attribute on the current GeoSeries must be set. Either `crs` or `epsg` may be specified for output.

This method will transform all points in all objects. It has no notion of 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.

## Parameters:

`crs : pyproj.CRS, optional if epsg is specified`

The value can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

`epsg : int, optional if crs is specified`

EPSG code specifying output projection.

`inplace : bool, optional, default: False`

Whether to return a new GeoDataFrame or do the transformation in place.

## Returns:

`GeoDataFrame`

## See also

`GeoDataFrame.set_crs`

assign CRS without re-projection

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d, crs=4326)
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
>>> gdf.crs
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
>>> gdf = gdf.to_crs(3857)
>>> gdf
 col1 geometry
0 name1 POINT (111319.491 222684.209)
1 name2 POINT (222638.982 111325.143)
>>> gdf.crs
<Projected CRS: EPSG:3857>
Name: WGS 84 / Pseudo-Mercator
Axis Info [cartesian]:
- X[east]: Easting (metre)
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.estimate\_utm\_crs

`GeoDataFrame.estimate_utm_crs(datum_name='WGS 84')`

[\[source\]](#)

Returns the estimated UTM CRS based on the bounds of the dataset.

 **Added in version 0.9.**

## Parameters:

`datum_name : str, optional`

The name of the datum to use in the query. Default is WGS 84.

## Returns:

`pypj.CRS`

## Examples

```
>>> import geodatasets
>>> df = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health")
...)
>>> df.estimate_utm_crs()
<Derived Projected CRS: EPSG:32616>
Name: WGS 84 / UTM zone 16N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Between 90°W and 84°W, northern hemisphere between equator and 84°N...
- bounds: (-90.0, 0.0, -84.0, 84.0)
Coordinate Operation:
- name: UTM zone 16N
- method: Transverse Mercator
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

# geopandas.GeoDataFrame.rename\_geometry

`GeoDataFrame.rename_geometry(col, inplace=False)`

[\[source\]](#)

Renames the GeoDataFrame geometry column to the specified name. By default yields a new object.

The original geometry column is replaced with the input.

## Parameters:

`col : new geometry column label`

`inplace : boolean, default False`

Modify the GeoDataFrame in place (do not create a new object)

## Returns:

`geodataframe : GeoDataFrame`

## See also

[`GeoDataFrame.set\_geometry`](#)

set the active geometry

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> df = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> df1 = df.rename_geometry('geom1')
>>> df1.geometry.name
'geom1'
>>> df.rename_geometry('geom1', inplace=True)
>>> df.geometry.name
'geom1'
```

# geopandas.GeoDataFrame.set\_geometry

`GeoDataFrame.set_geometry(col, drop=None, inplace=False, crs=None)`

[\[source\]](#)

Set the GeoDataFrame geometry using either an existing column or the specified input. By default yields a new object.

The original geometry column is replaced with the input.

## Parameters:

### `col : column label or array-like`

An existing column name or values to set as the new geometry column. If values (array-like, (Geo)Series) are passed, then if they are named (Series) the new geometry column will have the corresponding name, otherwise the existing geometry column will be replaced. If there is no existing geometry column, the new geometry column will use the default name “geometry”.

### `drop : boolean, default False`

When specifying a named Series or an existing column name for `col`, controls if the previous geometry column should be dropped from the result. The default of False keeps both the old and new geometry column.

! *Deprecated since version 1.0.0.*

### `inplace : boolean, default False`

Modify the GeoDataFrame in place (do not create a new object)

### `crs : pyproj.CRS, optional`

Coordinate system to use. The value can be anything accepted by

[`pyproj.CRS.from\_user\_input\(\)`](#), such as an authority string (eg “EPSG:4326”) or a WKT string. If passed, overrides both DataFrame and col’s crs. Otherwise, tries to get crs from passed col values or DataFrame.

## Returns:

`GeoDataFrame`

## See also

[`GeoDataFrame.rename\_geometry`](#)

rename an active geometry column

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

Passing an array:

```
>>> df1 = gdf.set_geometry([Point(0,0), Point(1,1)])
>>> df1
 col1 geometry
0 name1 POINT (0 0)
1 name2 POINT (1 1)
```

Using existing column:

```
>>> gdf["buffered"] = gdf.buffer(2)
>>> df2 = gdf.set_geometry("buffered")
>>> df2.geometry
0 POLYGON ((3 2, 2.99037 1.80397, 2.96157 1.6098...
1 POLYGON ((4 1, 3.99037 0.80397, 3.96157 0.6098...
Name: buffered, dtype: geometry
```

# geopandas.GeoDataFrame.active\_geometry

*property* `GeoDataFrame.active_geometry_name`

[\[source\]](#)

Return the name of the active geometry column

Returns a string name if a GeoDataFrame has an active geometry column set. Otherwise returns None. You can also access the active geometry column using the `.geometry` property. You can set a GeoSeries to be an active geometry using the [`set\_geometry\(\)`](#) method.

## Returns:

`str`

name of an active geometry column or None

## See also

[`GeoDataFrame.set\_geometry`](#)

set the active geometry

# geopandas.GeoDataFrame.dissolve

`GeoDataFrame.dissolve(by=None, aggfunc='first', as_index=True, level=None, sort=True, observed=False, dropna=True, method='unary', **kwargs)` [\[source\]](#)

Dissolve geometries within `groupby` into single observation. This is accomplished by applying the `union_all` method to all geometries within a groupself.

Observations associated with each `groupby` group will be aggregated using the `aggfunc`.

## Parameters:

### `by : str or list-like, default None`

Column(s) whose values define the groups to be dissolved. If None, the entire GeoDataFrame is considered as a single group. If a list-like object is provided, the values in the list are treated as categorical labels, and polygons will be combined based on the equality of these categorical labels.

### `aggfunc : function or string, default "first"`

Aggregation function for manipulation of data associated with each group. Passed to pandas `groupby.agg` method. Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. [np.sum, 'mean']
- dict of axis labels -> functions, function names or list of such.

### `as_index : boolean, default True`

If true, groupby columns become index of result.

### `level : int or str or sequence of int or sequence of str, default None`

If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

### `sort : bool, default True`

Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.

### `observed : bool, default False`

This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

### `dropna : bool, default True`

If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

### `method : str (default "unary")`

The method to use for the union. Options are:

- `"unary"`: use the unary union algorithm. This option is the most robust but can be slow for large numbers of geometries (default).
- `"coverage"`: use the coverage union algorithm. This option is optimized for non-overlapping polygons and can be significantly faster than the unary union algorithm. However, it can produce invalid geometries if the polygons overlap.

## \*\*kwargs

Keyword arguments to be passed to the pandas `DataFrameGroupby.agg` method which is used by `dissolve`. In particular, `numeric_only` may be supplied, which will be required in pandas 2.0 for certain aggfuncs.

 **Added in version 0.13.0.**

## Returns

—  
`GeoDataFrame`

### See also

[GeoDataFrame .explode](#)

explode multi-part geometries into single geometries

## Examples

```
>>> from shapely.geometry import Point
>>> d = {
... "col1": ["name1", "name2", "name1"],
... "geometry": [Point(1, 2), Point(2, 1), Point(0, 1)],
... }
>>> gdf = geopandas.GeoDataFrame(d, crs=4326)
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
2 name1 POINT (0 1)
```

```
>>> dissolved = gdf.dissolve('col1')
>>> dissolved
 geometry
col1
name1 MULTIPOINT ((0 1), (1 2))
name2 POINT (2 1)
```



# geopandas.GeoDataFrame.explode

`GeoDataFrame.explode(column=None, ignore_index=False, index_parts=False, **kwargs)`

[\[source\]](#)

Explode multi-part geometries into multiple single geometries.

Each row containing a multi-part geometry will be split into multiple rows with single geometries, thereby increasing the vertical size of the GeoDataFrame.

## Parameters:

### `column : string, default None`

Column to explode. In the case of a geometry column, multi-part geometries are converted to single-part. If None, the active geometry column is used.

### `ignore_index : bool, default False`

If True, the resulting index will be labelled 0, 1, ..., n - 1, ignoring `index_parts`.

### `index_parts : boolean, default False`

If True, the resulting index will be a multi-index (original index with an additional level indicating the multiple geometries: a new zero-based index for each single part geometry per multi-part geometry).

## Returns:

### `GeoDataFrame`

Exploded geodataframe with each single geometry as a separate entry in the geodataframe.

## See also

### [GeoDataFrame.dissolve](#)

dissolve geometries into a single observation.

## Examples

```
>>> from shapely.geometry import MultiPoint
>>> d = {
... "col1": ["name1", "name2"],
... "geometry": [
... MultiPoint([(1, 2), (3, 4)]),
... MultiPoint([(2, 1), (0, 0)]),
...],
... }
>>> gdf = geopandas.GeoDataFrame(d, crs=4326)
>>> gdf
 col1 geometry
0 name1 MULTIPOLYGON ((1 2, 3 4))
1 name2 MULTIPOLYGON ((2 1, 0 0))
```

```
>>> exploded = gdf.explode(index_parts=True)
>>> exploded
 col1 geometry
0 0 name1 POINT (1 2)
1 1 name1 POINT (3 4)
1 0 name2 POINT (2 1)
1 1 name2 POINT (0 0)
```

```
>>> exploded = gdf.explode(index_parts=False)
>>> exploded
 col1 geometry
0 name1 POINT (1 2)
0 name1 POINT (3 4)
1 name2 POINT (2 1)
1 name2 POINT (0 0)
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.sjoin

`GeoDataFrame.sjoin(df, *args, **kwargs)`

[\[source\]](#)

Spatial join of two GeoDataFrames.

See the User Guide page [Merging data](#) for details.

## Parameters:

**df : GeoDataFrame**

**how : string, default ‘inner’**

The type of join:

- ‘left’: use keys from left\_df; retain only left\_df geometry column
- ‘right’: use keys from right\_df; retain only right\_df geometry column
- ‘inner’: use intersection of keys from both dfs; retain only left\_df geometry column

**predicate : string, default ‘intersects’**

Binary predicate. Valid values are determined by the spatial index used. You can check the valid values in left\_df or right\_df as

`left_df.sindex.valid_query_predicates`

`right_df.sindex.valid_query_predicates`

**lsuffix : string, default ‘left’**

Suffix to apply to overlapping column names (left GeoDataFrame).

**rsuffix : string, default ‘right’**

Suffix to apply to overlapping column names (right GeoDataFrame).

**distance : number or array\_like, optional**

Distance(s) around each input geometry within which to query the tree for the ‘dwithin’ predicate. If array\_like, must be one-dimesional with length equal to length of left GeoDataFrame. Required if `predicate='dwithin'`.

**on\_attribute : string, list or tuple**

Column name(s) to join on as an additional join restriction on top of the spatial predicate.

These must be found in both DataFrames. If set, observations are joined only if the predicate applies and values in specified columns match.

## See also

[GeoDataFrame.sjoin\\_nearest](#)

nearest neighbor join

[sjoin](#)

equivalent top-level function

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> import geodatasets
>>> chicago = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_commpop"))
...)
>>> groceries = geopandas.read_file(
... geodatasets.get_path("geoda.groceries"))
...).to_crs(chicago.crs)
```

```
>>> chicago.head()
 community ... geometry
0 DOUGLAS ... MULTIPOLYGON (((-87.60914 41.84469, -87.60915 ...
1 OAKLAND ... MULTIPOLYGON (((-87.59215 41.81693, -87.59231 ...
2 FULLER PARK ... MULTIPOLYGON (((-87.62880 41.80189, -87.62879 ...
3 GRAND BOULEVARD ... MULTIPOLYGON (((-87.60671 41.81681, -87.60670 ...
4 KENWOOD ... MULTIPOLYGON (((-87.59215 41.81693, -87.59215 ...
```

[5 rows x 9 columns]

```
>>> groceries.head()
 OBJECTID Ycoord ... Category geometry
0 16 41.973266 ... NaN MULTIPOLY(((-87.65661 41.97321))
1 18 41.696367 ... NaN MULTIPOLY(((-87.68136 41.69713))
2 22 41.868634 ... NaN MULTIPOLY(((-87.63918 41.86847))
3 23 41.877590 ... new MULTIPOLY(((-87.65495 41.87783))
4 27 41.737696 ... NaN MULTIPOLY(((-87.62715 41.73623))
```

```
>>> groceries_w_communities = groceries.sjoin(chicago)
>>> groceries_w_communities[["OBJECTID", "community", "geometry"]].head()
 OBJECTID community geometry
0 16 UPTOWN MULTIPOLY(((-87.65661 41.97321))
1 18 MORGAN PARK MULTIPOLY(((-87.68136 41.69713))
2 22 NEAR WEST SIDE MULTIPOLY(((-87.63918 41.86847))
3 23 NEAR WEST SIDE MULTIPOLY(((-87.65495 41.87783))
4 27 CHATHAM MULTIPOLY(((-87.62715 41.73623))
```

# geopandas.GeoDataFrame.sjoin\_nearest

```
GeoDataFrame.sjoin_nearest(right, how='inner', max_distance=None,
lsuffix='left', rsuffix='right', distance_col=None, exclusive=False) [source]
```

Spatial join of two GeoDataFrames based on the distance between their geometries.

Results will include multiple output records for a single input record where there are multiple equidistant nearest or intersected neighbors.

See the User Guide page

[https://geopandas.readthedocs.io/en/latest/docs/user\\_guide/mergingdata.html](https://geopandas.readthedocs.io/en/latest/docs/user_guide/mergingdata.html) for more details.

## Parameters:

**right : GeoDataFrame**

**how : string, default ‘inner’**

The type of join:

- ‘left’: use keys from left\_df; retain only left\_df geometry column
- ‘right’: use keys from right\_df; retain only right\_df geometry column
- ‘inner’: use intersection of keys from both dfs; retain only left\_df geometry column

**max\_distance : float, default None**

Maximum distance within which to query for nearest geometry. Must be greater than 0. The max\_distance used to search for nearest items in the tree may have a significant impact on performance by reducing the number of input geometries that are evaluated for nearest items in the tree.

**lsuffix : string, default ‘left’**

Suffix to apply to overlapping column names (left GeoDataFrame).

**rsuffix : string, default ‘right’**

Suffix to apply to overlapping column names (right GeoDataFrame).

**distance\_col : string, default None**

If set, save the distances computed between matching geometries under a column of this name in the joined GeoDataFrame.

**exclusive : bool, optional, default False**

If True, the nearest geometries that are equal to the input geometry will not be returned, default False. Requires Shapely >= 2.0

## See also

[GeoDataFrame.sjoin](#)

binary predicate joins

[sjoin\\_nearest](#)

equivalent top-level function

## Notes

Since this join relies on distances, results will be inaccurate if your geometries are in a geographic CRS.

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> import geodatasets
>>> groceries = geopandas.read_file(
... geodatasets.get_path("geoda.groceries")
...)
>>> chicago = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health")
...).to_crs(groceries.crs)
```

```
>>> chicago.head()
ComAreaID ... geometry
0 35 ... POLYGON ((-87.60914 41.84469, -87.60915 41.844...
1 36 ... POLYGON ((-87.59215 41.81693, -87.59231 41.816...
2 37 ... POLYGON ((-87.62880 41.80189, -87.62879 41.801...
3 38 ... POLYGON ((-87.60671 41.81681, -87.60670 41.816...
4 39 ... POLYGON ((-87.59215 41.81693, -87.59215 41.816...
[5 rows x 87 columns]
```

```
>>> groceries.head()
OBJECTID Ycoord ... Category geometry
0 16 41.973266 ... NaN MULTIPOLY ((-87.65661 41.97321))
1 18 41.696367 ... NaN MULTIPOLY ((-87.68136 41.69713))
2 22 41.868634 ... NaN MULTIPOLY ((-87.63918 41.86847))
3 23 41.877590 ... new MULTIPOLY ((-87.65495 41.87783))
4 27 41.737696 ... NaN MULTIPOLY ((-87.62715 41.73623))
[5 rows x 8 columns]
```

```
>>> groceries_w_communities = groceries.sjoin_nearest(chicago)
>>> groceries_w_communities[["Chain", "community", "geometry"]].head(2)
 Chain community geometry
0 VIET HOA PLAZA UPTOWN MULTIPOLY ((1168268.672 1933554.35))
1 COUNTY FAIR FOODS MORGAN PARK MULTIPOLY ((1162302.618 1832900.224))
```

To include the distances:

```
>>> groceries_w_communities = groceries.sjoin_nearest(chicago, distance_col="distances")
>>> groceries_w_communities[["Chain", "community", "distances"]].head(2)
 Chain community distances
0 VIET HOA PLAZA UPTOWN 0.0
1 COUNTY FAIR FOODS MORGAN PARK 0.0
```

In the following example, we get multiple groceries for Uptown because all results are equidistant (in this case zero because they intersect). In fact, we get 4 results in total:

```
>>> chicago_w_groceries = groceries.sjoin_nearest(chicago, distance_col="distances",
>>> uptown_results = chicago_w_groceries[chicago_w_groceries["community"] == "UPTOWN"]
>>> uptown_results[["Chain", "community"]]
 Chain community
30 VIET HOA PLAZA UPTOWN
30 JEWEL OSCO UPTOWN
30 TARGET UPTOWN
30 Mariano's UPTOWN
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.clip

`GeoDataFrame.clip(mask, keep_geom_type=False, sort=False)`

[\[source\]](#)

Clip points, lines, or polygon geometries to the mask extent.

Both layers must be in the same Coordinate Reference System (CRS). The GeoDataFrame will be clipped to the full extent of the `mask` object.

If there are multiple polygons in mask, data from the GeoDataFrame will be clipped to the total boundary of all polygons in mask.

## Parameters:

**mask : *GeoDataFrame, GeoSeries, (Multi)Polygon, list-like***

Polygon vector layer used to clip the GeoDataFrame. The mask's geometry is dissolved into one geometric feature and intersected with GeoDataFrame. If the mask is list-like with four elements `(minx, miny, maxx, maxy)`, `clip` will use a faster rectangle clipping ([`clip\_by\_rect\(\)`](#)), possibly leading to slightly different results.

**keep\_geom\_type : *boolean, default False***

If True, return only geometries of original type in case of intersection resulting in multiple geometry types or GeometryCollections. If False, return all resulting geometries (potentially mixed types).

**sort : *boolean, default False***

If True, the order of rows in the clipped GeoDataFrame will be preserved at small performance cost. If False the order of rows in the clipped GeoDataFrame will be random.

## Returns:

**GeoDataFrame**

Vector data (points, lines, polygons) from the GeoDataFrame clipped to polygon boundary from mask.

## See also

[`clip`](#)

equivalent top-level function

## Examples

Clip points (grocery stores) with polygons (the Near West Side community):

```
>>> import geodatasets
>>> chicago = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health"))
...)
>>> near_west_side = chicago[chicago["community"] == "NEAR WEST SIDE"]
>>> groceries = geopandas.read_file(
... geodatasets.get_path("geoda.groceries"))
...).to_crs(chicago.crs)
>>> groceries.shape
(148, 8)
```

```
>>> nws_groceries = groceries.clip(near_west_side)
>>> nws_groceries.shape
(7, 8)
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.overlay

`GeoDataFrame.overlay(right, how='intersection', keep_geom_type=None, make_valid=True)`

[\[source\]](#)

Perform spatial overlay between GeoDataFrames.

Currently only supports data GeoDataFrames with uniform geometry types, i.e. containing only (Multi)Polygons, or only (Multi)Points, or a combination of (Multi)LineString and LinearRing shapes. Implements several methods that are all effectively subsets of the union.

See the User Guide page [Set operations with overlay](#) for details.

## Parameters:

**right : `GeoDataFrame`**

**how : `string`**

Method of spatial overlay: ‘intersection’, ‘union’, ‘identity’, ‘symmetric\_difference’ or ‘difference’.

**keep\_geom\_type : `bool`**

If True, return only geometries of the same geometry type the GeoDataFrame has, if False, return all resulting geometries. Default is None, which will set `keep_geom_type` to True but warn upon dropping geometries.

**make\_valid : `bool, default True`**

If True, any invalid input geometries are corrected with a call to `make_valid()`, if False, a `ValueError` is raised if any input geometries are invalid.

## Returns:

**df : `GeoDataFrame`**

GeoDataFrame with new set of polygons and attributes resulting from the overlay

## See also

[`GeoDataFrame.sjoin`](#)

spatial join

[`overlay`](#)

equivalent top-level function

## Notes

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> from shapely.geometry import Polygon
>>> polys1 = geopandas.GeoSeries([Polygon([(0,0), (2,0), (2,2), (0,2)]),
... Polygon([(2,2), (4,2), (4,4), (2,4)])])
>>> polys2 = geopandas.GeoSeries([Polygon([(1,1), (3,1), (3,3), (1,3)]),
... Polygon([(3,3), (5,3), (5,5), (3,5)])])
>>> df1 = geopandas.GeoDataFrame({'geometry': polys1, 'df1_data':[1,2]})
>>> df2 = geopandas.GeoDataFrame({'geometry': polys2, 'df2_data':[1,2]})
```

```
>>> df1.overlay(df2, how='union')
 df1_data df2_data geometry
0 1.0 1.0 POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))
1 2.0 1.0 POLYGON ((2 2, 2 3, 3 3, 3 2, 2 2))
2 2.0 2.0 POLYGON ((4 4, 4 3, 3 3, 3 4, 4 4))
3 1.0 NaN POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))
4 2.0 NaN MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4...
5 NaN 1.0 MULTIPOLYGON (((2 3, 2 2, 1 2, 1 3, 2 3)), ((3...
6 NaN 2.0 POLYGON ((3 5, 5 5, 5 3, 4 3, 4 4, 3 4, 3 5))
```

```
>>> df1.overlay(df2, how='intersection')
 df1_data df2_data geometry
0 1.0 1 POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))
1 2.0 1 POLYGON ((2 2, 2 3, 3 3, 3 2, 2 2))
2 2.0 2 POLYGON ((4 4, 4 3, 3 3, 3 4, 4 4))
```

```
>>> df1.overlay(df2, how='symmetric_difference')
 df1_data df2_data geometry
0 1.0 NaN POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))
1 2.0 NaN MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4...
2 NaN 1.0 MULTIPOLYGON (((2 3, 2 2, 1 2, 1 3, 2 3)), ((3...
3 NaN 2.0 POLYGON ((3 5, 5 5, 5 3, 4 3, 4 4, 3 4, 3 5))
```

```
>>> df1.overlay(df2, how='difference')
 geometry df1_data
0 POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0)) 1
1 MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4... 2
```

```
>>> df1.overlay(df2, how='identity')
 df1_data df2_data geometry
0 1.0 1.0 POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))
1 2.0 1.0 POLYGON ((2 2, 2 3, 3 3, 3 2, 2 2))
2 2.0 2.0 POLYGON ((4 4, 4 3, 3 3, 3 4, 4 4))
3 1.0 NaN POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))
4 2.0 NaN MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4...
```



# geopandas.GeoDataFrame.explore

`GeoDataFrame.explore(*args, **kwargs)`

[\[source\]](#)

Interactive map based on GeoPandas and folium/leaflet.js

Generate an interactive leaflet map based on `GeoDataFrame`

## Parameters:

**column : str, np.array, pd.Series (default None)**

The name of the dataframe column, `numpy.array`, or `pandas.Series` to be plotted. If `numpy.array` or `pandas.Series` are used then it must have same length as dataframe.

**cmap : str, matplotlib.Colormap, branca.colormap or function (default None)**

The name of a colormap recognized by `matplotlib`, a list-like of colors, `matplotlib.colors.Colormap`, a `branca.colormap.ColorMap` or function that returns a named color or hex based on the column value, e.g.:

```
def my_colormap(value): # scalar value defined in 'column'
 if value > 1:
 return "green"
 return "red"
```

**color : str, array-like (default None)**

Named color or a list-like of colors (named or hex).

**m : folium.Map (default None)**

Existing map instance on which to draw the plot.

**tiles : str, xyzservices.TileProvider (default ‘OpenStreetMap Mapnik’)**

Map tileset to use. Can choose from the list supported by folium, query a `xyzservices.TileProvider` by a name from `xyzservices.providers`, pass `xyzservices.TileProvider` object or pass custom XYZ URL. The current list of built-in providers (when `xyzservices` is not available):

```
["OpenStreetMap", "CartoDB positron", "CartoDB dark_matter"]
```

You can pass a custom tileset to Folium by passing a Leaflet-style URL to the tiles parameter: `http://{s}.yourtiles.com/{z}/{x}/{y}.png`. Be sure to check their terms and conditions and to provide attribution with the `attr` keyword.

**attr : str (default None)**

Map tile attribution; only required if passing custom tile URL.

**tooltip : bool, str, int, list (default True)**

Display GeoDataFrame attributes when hovering over the object. `True` includes all columns. `False` removes tooltip. Pass string or list of strings to specify a column(s). Integer

specifies first n columns to be included. Defaults to `True`.

### **popup : bool, str, int, list (default False)**

Input GeoDataFrame attributes for object displayed when clicking. `True` includes all columns. `False` removes popup. Pass string or list of strings to specify a column(s). Integer specifies first n columns to be included. Defaults to `False`.

### **highlight : bool (default True)**

Enable highlight functionality when hovering over a geometry.

### **categorical : bool (default False)**

If `False`, `cmap` will reflect numerical values of the column being plotted. For non-numerical columns, this will be set to True.

### **legend : bool (default True)**

Plot a legend in choropleth plots. Ignored if no `column` is given.

### **scheme : str (default None)**

Name of a choropleth classification scheme (requires `mapclassify` >= 2.4.0). A `mapclassify.classify()` will be used under the hood. Supported are all schemes provided by `mapclassify` (e.g. `'BoxPlot'`, `'EqualInterval'`, `'FisherJenks'`, `'FisherJenksSampled'`, `'HeadTailBreaks'`, `'JenksCaspall'`, `'JenksCaspallForced'`, `'JenksCaspallSampled'`, `'MaxP'`, `'MaximumBreaks'`, `'NaturalBreaks'`, `'Quantiles'`, `'Percentiles'`, `'StdMean'`, `'UserDefined'`). Arguments can be passed in `classification_kwds`.

### **k : int (default 5)**

Number of classes

### **vmin : None or float (default None)**

Minimum value of `cmap`. If `None`, the minimum data value in the column to be plotted is used.

### **vmax : None or float (default None)**

Maximum value of `cmap`. If `None`, the maximum data value in the column to be plotted is used.

### **width : pixel int or percentage string (default: '100%')**

Width of the folium `Map`. If the argument m is given explicitly, width is ignored.

### **height : pixel int or percentage string (default: '100%')**

Height of the folium `Map`. If the argument m is given explicitly, height is ignored.

### **categories : list-like**

Ordered list-like object of categories to be used for categorical plot.

### **classification\_kwds : dict (default None)**

Keyword arguments to pass to mapclassify

**control\_scale : bool, (default True)**

Whether to add a control scale on the map.

**marker\_type : str, folium.Circle, folium.CircleMarker, folium.Marker (default None)**

Allowed string options are ('marker', 'circle', 'circle\_marker'). Defaults to folium.CircleMarker.

**marker\_kwds: dict (default {})**

Additional keywords to be passed to the selected `marker_type`, e.g.:

**radius : float (default 2 for circle\_marker and 50 for circle))**

Radius of the circle, in meters (for `circle`) or pixels (for `circle_marker`).

**fill : bool (default True)**

Whether to fill the `circle` or `circle_marker` with color.

**icon : folium.map.Icon**

the `folium.map.Icon` object to use to render the marker.

**draggable : bool (default False)**

Set to True to be able to drag the marker around the map.

**style\_kwds : dict (default {})**

Additional style to be passed to folium `style_function`:

**stroke : bool (default True)**

Whether to draw stroke along the path. Set it to `False` to disable borders on polygons or circles.

**color : str**

Stroke color

**weight : int**

Stroke width in pixels

**opacity : float (default 1.0)**

Stroke opacity

**fill : boolean (default True)**

Whether to fill the path with color. Set it to `False` to disable filling on polygons or circles.

**fillColor : str**

Fill color. Defaults to the value of the color option

**fillOpacity : float (default 0.5)**

Fill opacity.

**style\_function : callable**

Function mapping a GeoJson Feature to a style `dict`.

- Style properties `folium.vector_layers.path_options()`

- GeoJson features [GeoDataFrame.\\_\\_geo\\_interface\\_\\_](#)

e.g.:

```
lambda x: {"color": "red" if x["properties"]["gdp_md_est"]<10**6
 else "blue"}
```

Plus all supported by [folium.vector\\_layers.path\\_options\(\)](#). See the documentation of [folium.features.GeoJson](#) for details.

### **highlight\_kwds : dict (default {})**

Style to be passed to folium highlight\_function. Uses the same keywords as [style\\_kwds](#).

When empty, defaults to `{"fillOpacity": 0.75}`.

### **tooltip\_kwds : dict (default {})**

Additional keywords to be passed to [folium.features.GeoJsonTooltip](#), e.g. [aliases](#), [labels](#), or [sticky](#).

### **popup\_kwds : dict (default {})**

Additional keywords to be passed to [folium.features.GeoJsonPopup](#), e.g. [aliases](#) or [labels](#).

### **legend\_kwds : dict (default {})**

Additional keywords to be passed to the legend.

Currently supported customisation:

#### **caption : string**

Custom caption of the legend. Defaults to the column name.

Additional accepted keywords when [scheme](#) is specified:

#### **colorbar : bool (default True)**

An option to control the style of the legend. If True, continuous colorbar will be used. If False, categorical legend will be used for bins.

#### **scale : bool (default True)**

Scale bins along the colorbar axis according to the bin edges (True) or use the equal length for each bin (False)

#### **fmt : string (default “{:2f}”)**

A formatting specification for the bin edges of the classes in the legend. For example, to have no decimals: `{"fmt": "{:.0f}"}`. Applies if [colorbar=False](#).

#### **labels : list-like**

A list of legend labels to override the auto-generated labels. Needs to have the same number of elements as the number of classes ( $k$ ). Applies if [colorbar=False](#).

#### **interval : boolean (default False)**

An option to control brackets from mapclassify legend. If True, open/closed interval brackets are shown in the legend. Applies if `colorbar=False`.

### **max\_labels : int, default 10**

Maximum number of colorbar tick labels (requires branca>=0.5.0)

### **map\_kwds : dict (default {})**

Additional keywords to be passed to folium `Map`, e.g. `dragging`, or `scrollWheelZoom`.

### **\*\*kwargs : dict**

Additional options to be passed on to the folium object.

## Returns:

**m : folium.folium.Map**

folium `Map` instance

## Examples

```
>>> import geodatasets
>>> df = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health")
...)
>>> df.head(2)
ComAreaID ... geometry
0 35 ... POLYGON ((-87.60914 41.84469, -87.60915 41.844...
1 36 ... POLYGON ((-87.59215 41.81693, -87.59231 41.816...
```

[2 rows x 87 columns]

```
>>> df.explore("Pop2012", cmap="Blues")
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.GeoDataFrame.plot

## GeoDataFrame.plot()

[\[source\]](#)

Plot a GeoDataFrame.

Generate a plot of a GeoDataFrame with matplotlib. If a column is specified, the plot coloring will be based on values in that column.

### Parameters:

#### column : str, np.array, pd.Series (default None)

The name of the dataframe column, np.array, or pd.Series to be plotted. If np.array or pd.Series are used then it must have same length as dataframe. Values are used to color the plot. Ignored if *color* is also set.

#### kind: str

The kind of plots to produce. The default is to create a map (“geo”). Other supported kinds of plots from pandas:

- ‘line’ : line plot
- ‘bar’ : vertical bar plot
- ‘barh’ : horizontal bar plot
- ‘hist’ : histogram
- ‘box’ : BoxPlot
- ‘kde’ : Kernel Density Estimation plot
- ‘density’ : same as ‘kde’
- ‘area’ : area plot
- ‘pie’ : pie plot
- ‘scatter’ : scatter plot
- ‘hexbin’ : hexbin plot.

#### cmap : str (default None)

The name of a colormap recognized by matplotlib.

#### color : str, np.array, pd.Series (default None)

If specified, all objects will be colored uniformly.

#### ax : matplotlib.pyplot.Artist (default None)

axes on which to draw the plot

#### cax : matplotlib.pyplot Artist (default None)

axes on which to draw the legend in case of color map.

#### categorical : bool (default False)

If False, cmap will reflect numerical values of the column being plotted. For non-numerical columns, this will be set to True.

### **legend : bool (default False)**

Plot a legend. Ignored if no *column* is given, or if *color* is given.

### **scheme : str (default None)**

Name of a choropleth classification scheme (requires mapclassify). A mapclassify.MapClassifier object will be used under the hood. Supported are all schemes provided by mapclassify (e.g. ‘BoxPlot’, ‘EqualInterval’, ‘FisherJenks’, ‘FisherJenksSampled’, ‘HeadTailBreaks’, ‘JenksCaspall’, ‘JenksCaspallForced’, ‘JenksCaspallSampled’, ‘MaxP’, ‘MaximumBreaks’, ‘NaturalBreaks’, ‘Quantiles’, ‘Percentiles’, ‘StdMean’, ‘UserDefined’). Arguments can be passed in classification\_kwds.

### **k : int (default 5)**

Number of classes (ignored if scheme is None)

### **vmin : None or float (default None)**

Minimum value of cmap. If None, the minimum data value in the column to be plotted is used.

### **vmax : None or float (default None)**

Maximum value of cmap. If None, the maximum data value in the column to be plotted is used.

### **markersize : str or float or sequence (default None)**

Only applies to point geometries within a frame. If a str, will use the values in the column of the frame specified by markersize to set the size of markers. Otherwise can be a value to apply to all points, or a sequence of the same length as the number of points.

### **figsize : tuple of integers (default None)**

Size of the resulting matplotlib.figure.Figure. If the argument axes is given explicitly, figsize is ignored.

### **legend\_kwds : dict (default None)**

Keyword arguments to pass to [`matplotlib.pyplot.legend\(\)`](#) or

[`matplotlib.pyplot.colorbar\(\)`](#). Additional accepted keywords when *scheme* is specified:

### **fmt : string**

A formatting specification for the bin edges of the classes in the legend. For example, to have no decimals: `{"fmt": "{:.0f}"}`.

### **labels : list-like**

A list of legend labels to override the auto-generated labels. Needs to have the same number of elements as the number of classes (*k*).

### **interval : boolean (default False)**

An option to control brackets from mapclassify legend. If True, open/closed interval brackets are shown in the legend.

### categories : *list-like*

Ordered list-like object of categories to be used for categorical plot.

### classification\_kwds : *dict (default None)*

Keyword arguments to pass to mapclassify

### missing\_kwds : *dict (default None)*

Keyword arguments specifying color options (as style\_kwds) to be passed on to geometries with missing values in addition to or overwriting other style kwds. If None, geometries with missing values are not plotted.

### aspect : ‘auto’, ‘equal’, *None* or *float* (default ‘auto’)

Set aspect of axis. If ‘auto’, the default aspect for map plots is ‘equal’; if however data are not projected (coordinates are long/lat), the aspect is by default set to  $1/\cos(df_y * \pi/180)$  with df\_y the y coordinate of the middle of the GeoDataFrame (the mean of the y range of bounding box) so that a long/lat square appears square in the middle of the plot. This implies an Equirectangular projection. If None, the aspect of ax won’t be changed. It can also be set manually (float) as the ratio of y-unit to x-unit.

### autolim : *bool (default True)*

Update axes data limits to contain the new geometries.

### \*\*style\_kwds : *dict*

Style options to be passed on to the actual plot function, such as `edgecolor`, `facecolor`, `linewidth`, `markersize`, `alpha`.

## Returns:

`ax` : *matplotlib axes instance*

## Examples

```
>>> import geodatasets
>>> df = geopandas.read_file(geodatasets.get_path("nybb"))
>>> df.head()
 BoroCode ... geometry
0 5 ... MULTIPOLYGON (((970217.022 145643.332, 970227...
1 4 ... MULTIPOLYGON (((1029606.077 156073.814, 102957...
2 3 ... MULTIPOLYGON (((1021176.479 151374.797, 102100...
3 1 ... MULTIPOLYGON (((981219.056 188655.316, 980940...
4 2 ... MULTIPOLYGON (((1012821.806 229228.265, 101278...
```

```
>>> df.plot("BoroName", cmap="Set1")
```

See the User Guide page [Mapping and plotting tools](#) for details.



# geopandas.GeoDataFrame.sindex

**property** `GeoDataFrame.sindex`

[\[source\]](#)

Generate the spatial index

Creates R-tree spatial index based on `shapely.STRtree`.

Note that the spatial index may not be fully initialized until the first use.

## Examples

```
>>> from shapely.geometry import box
>>> s = geopandas.GeoSeries(geopandas.points_from_xy(range(5), range(5)))
>>> s
0 POINT (0 0)
1 POINT (1 1)
2 POINT (2 2)
3 POINT (3 3)
4 POINT (4 4)
dtype: geometry
```

Query the spatial index with a single geometry based on the bounding box:

```
>>> s.sindex.query(box(1, 1, 3, 3))
array([1, 2, 3])
```

Query the spatial index with a single geometry based on the predicate:

```
>>> s.sindex.query(box(1, 1, 3, 3), predicate="contains")
array([2])
```

Query the spatial index with an array of geometries based on the bounding box:

```
>>> s2 = geopandas.GeoSeries([box(1, 1, 3, 3), box(4, 4, 5, 5)])
>>> s2
0 POLYGON ((3 1, 3 3, 1 3, 1 1, 3 1))
1 POLYGON ((5 4, 5 5, 4 5, 4 4, 5 4))
dtype: geometry
```

```
>>> s.sindex.query(s2)
array([[0, 0, 0, 1],
 [1, 2, 3, 4]])
```

Query the spatial index with an array of geometries based on the predicate:

```
>>> s.sindex.query(s2, predicate="contains")
array([[0],
 [2]])
```

>>>

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.GeoDataFrame.has\_sindex

**property** `GeoDataFrame.has_sindex`

[\[source\]](#)

Check the existence of the spatial index without generating it.

Use the `.sindex` attribute on a GeoDataFrame or GeoSeries to generate a spatial index if it does not yet exist, which may take considerable time based on the underlying index implementation.

Note that the underlying spatial index may not be fully initialized until the first use.

## Returns:

`bool`

*True* if the spatial index has been generated or *False* if not.

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d)
>>> gdf.has_sindex
False
>>> index = gdf.sindex
>>> gdf.has_sindex
True
```

# geopandas.GeoDataFrame.cx

## property GeoDataFrame.cx

[\[source\]](#)

Coordinate based indexer to select by intersection with bounding box.

Format of input should be `.cx[xmin:xmax, ymin:ymax]`. Any of `xmin`, `xmax`, `ymin`, and `ymax` can be provided, but input must include a comma separating x and y slices. That is, `.cx[:, :]` will return the full series/frame, but `.cx[:]` is not implemented.

## Examples

```
>>> from shapely.geometry import LineString, Point
>>> s = geopandas.GeoSeries(
... [Point(0, 0), Point(1, 2), Point(3, 3), LineString([(0, 0), (3, 3)])]
...)
>>> s
0 POINT (0 0)
1 POINT (1 2)
2 POINT (3 3)
3 LINESTRING (0 0, 3 3)
dtype: geometry
```

```
>>> s.cx[0:1, 0:1]
0 POINT (0 0)
3 LINESTRING (0 0, 3 3)
dtype: geometry
```

```
>>> s.cx[:, 1:]
1 POINT (1 2)
2 POINT (3 3)
3 LINESTRING (0 0, 3 3)
dtype: geometry
```

# geopandas.GeoDataFrame.\_\_geo\_interface\_\_

**property** `GeoDataFrame.__geo_interface__`

[\[source\]](#)

Returns a `GeoDataFrame` as a python feature collection.

Implements the `geo_interface`. The returned python data structure represents the `GeoDataFrame` as a GeoJSON-like `FeatureCollection`.

This differs from `to_geo_dict()` only in that it is a property with default args instead of a method.

CRS of the dataframe is not passed on to the output, unlike `to_json()`.

## Examples

```
>>> >>>
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

```
>>> >>>
>>> gdf.__geo_interface__
{'type': 'FeatureCollection', 'features': [{"id": "0", "type": "Feature", "properties":
```

# geopandas.GeoDataFrame.iterfeatures

`GeoDataFrame.iterfeatures(na='null', show_bbox=False, drop_id=False)` [\[source\]](#)

Returns an iterator that yields feature dictionaries that comply with `__geo_interface__`

## Parameters:

`na : str, optional`

Options are {'null', 'drop', 'keep'}, default 'null'. Indicates how to output missing (NaN) values in the GeoDataFrame

- null: output the missing entries as JSON null
- drop: remove the property from the feature. This applies to each feature individually so that features may have different properties
- keep: output the missing entries as NaN

`show_bbox : bool, optional`

Include bbox (bounds) in the geojson. Default False.

`drop_id : bool, default: False`

Whether to retain the index of the GeoDataFrame as the id property in the generated GeoJSON. Default is False, but may want True if the index is just arbitrary row numbers.

## Examples

```
>>> from shapely.geometry import Point
>>> d = {'col1': ['name1', 'name2'], 'geometry': [Point(1, 2), Point(2, 1)]}
>>> gdf = geopandas.GeoDataFrame(d, crs="EPSG:4326")
>>> gdf
 col1 geometry
0 name1 POINT (1 2)
1 name2 POINT (2 1)
```

```
>>> feature = next(gdf.iterfeatures())
>>> feature
{'id': '0', 'type': 'Feature', 'properties': {'col1': 'name1'}, 'geometry': {'type': :
```

# Input/output

## GIS vector files

<code>list_layers</code> (filename)	List layers available in a file.
<code>read_file</code> (filename[, bbox, mask, columns, ...])	Returns a GeoDataFrame from a file or URL.
<code>GeoDataFrame.to_file</code> (filename[, driver, ...])	Write the <code>GeoDataFrame</code> to a file.

## PostGIS

<code>read_postgis</code> (sql, con[, geom_col, crs, ...])	Returns a GeoDataFrame corresponding to the result of the query string, which must contain a geometry column in WKB representation.
<code>GeoDataFrame.to_postgis</code> (name, con[, schema, ...])	Upload GeoDataFrame into PostGIS database.

## Feather

<code>read_feather</code> (path[, columns])	Load a Feather object from the file path, returning a GeoDataFrame.
<code>GeoDataFrame.to_feather</code> (path[, index, ...])	Write a GeoDataFrame to the Feather format.

## Parquet

<code>read_parquet</code> (path[, columns, ...])	Load a Parquet object from the file path, returning a GeoDataFrame.
<code>GeoDataFrame.to_parquet</code> (path[, index, ...])	Write a GeoDataFrame to the Parquet format.

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.list\_layers

`geopandas.list_layers(filename)`

[\[source\]](#)

List layers available in a file.

Provides an overview of layers available in a file or URL together with their geometry types. When supported by the data source, this includes both spatial and non-spatial layers. Non-spatial layers are indicated by the `"geometry_type"` column being `None`. GeoPandas will not read such layers but they can be read into a pd.DataFrame using [`pyogrio.read\_dataframe\(\)`](#).

## Parameters:

`filename : str, path object or file-like object`

Either the absolute or relative path to the file or URL to be opened, or any object with a `read()` method (such as an open file or StringIO)

## Returns:

`pandas.DataFrame`

A DataFrame with columns “name” and “geometry\_type” and one row per layer.

# geopandas.read\_file

`geopandas.read_file(filename, bbox=None, mask=None, columns=None, rows=None, engine=None, **kwargs)`

[\[source\]](#)

Returns a GeoDataFrame from a file or URL.

## Parameters:

### **filename : str, path object or file-like object**

Either the absolute or relative path to the file or URL to be opened, or any object with a `read()` method (such as an open file or `StringIO`)

### **bbox : tuple | GeoDataFrame or GeoSeries | shapely Geometry, default None**

Filter features by given bounding box, `GeoSeries`, `GeoDataFrame` or a `shapely geometry`.

With `engine="fiona"`, CRS mis-matches are resolved if given a `GeoSeries` or `GeoDataFrame`.

With `engine="pyogrio"`, `bbox` must be in the same CRS as the dataset. Tuple is (`minx`, `miny`, `maxx`, `maxy`) to match the `bounds` property of `shapely geometry` objects. Cannot be used with `mask`.

### **mask : dict | GeoDataFrame or GeoSeries | shapely Geometry, default None**

Filter for features that intersect with the given dict-like `geojson geometry`, `GeoSeries`, `GeoDataFrame` or `shapely geometry`. CRS mis-matches are resolved if given a `GeoSeries` or `GeoDataFrame`. Cannot be used with `bbox`. If multiple geometries are passed, this will first union all geometries, which may be computationally expensive.

### **columns : list, optional**

List of column names to import from the data source. Column names must exactly match the names in the data source. To avoid reading any columns (besides the geometry column), pass an empty list-like. By default reads all columns.

### **rows : int or slice, default None**

Load in specific rows by passing an integer (first  $n$  rows) or a `slice()` object.

### **engine : str, "pyogrio" or "fiona"**

The underlying library that is used to read the file. Currently, the supported options are "pyogrio" and "fiona". Defaults to "pyogrio" if installed, otherwise tries "fiona". Engine can also be set globally with the `geopandas.options.io_engine` option.

### **\*\*kwargs**

Keyword args to be passed to the engine, and can be used to write to multi-layer data, store data within archives (zip files), etc. In case of the "pyogrio" engine, the keyword arguments are passed to `pyogrio.write_dataframe`. In case of the "fiona" engine, the keyword arguments are passed to `fiona.open``. For more information on possible keywords, type: `import pyogrio; help(pyogrio.write_dataframe)`.

## Returns:

`geopandas.GeoDataFrame` or `pandas.DataFrame`

If `ignore_geometry=True` a `pandas.DataFrame` will be returned.

## Notes

The format drivers will attempt to detect the encoding of your data, but may fail. In this case, the proper encoding can be specified explicitly by using the `encoding` keyword parameter, e.g.

`encoding='utf-8'`.

When specifying a URL, geopandas will check if the server supports reading partial data and in that case pass the URL as is to the underlying engine, which will then use the network file system handler of GDAL to read from the URL. Otherwise geopandas will download the data from the URL and pass all data in-memory to the underlying engine. If you need more control over how the URL is read, you can specify the GDAL virtual filesystem manually (e.g. `/vsicurl/https://...`). See the GDAL documentation on filesystems for more details ([https://gdal.org/user/virtual\\_file\\_systems.html#vsicurl-http-https-ftp-files-random-access](https://gdal.org/user/virtual_file_systems.html#vsicurl-http-https-ftp-files-random-access)).

## Examples

```
>>> df = geopandas.read_file("nybb.shp")
```

Specifying layer of GPKG:

```
>>> df = geopandas.read_file("file.gpkg", layer='cities')
```

Reading only first 10 rows:

```
>>> df = geopandas.read_file("nybb.shp", rows=10)
```

Reading only geometries intersecting `mask`:

```
>>> df = geopandas.read_file("nybb.shp", mask=polygon)
```

Reading only geometries intersecting `bbox`:

```
>>> df = geopandas.read_file("nybb.shp", bbox=(0, 0, 10, 20))
```



# geopandas.read\_postgis

`geopandas.read_postgis(sql, con, geom_col='geom', crs=None, index_col=None, coerce_float=True, parse_dates=None, params=None, chunksize=None)` [\[source\]](#)

Returns a GeoDataFrame corresponding to the result of the query string, which must contain a geometry column in WKB representation.

It is also possible to use [read\\_file\(\)](#) to read from a database. Especially for file geodatabases like GeoPackage or SpatiaLite this can be easier.

## Parameters:

### `sql : string`

SQL query to execute in selecting entries from database, or name of the table to read from the database.

### `con : sqlalchemy.engine.Connection or sqlalchemy.engine.Engine`

Active connection to the database to query.

### `geom_col : string, default 'geom'`

column name to convert to shapely geometries

### `crs : dict or str, optional`

CRS to use for the returned GeoDataFrame; if not set, tries to determine CRS from the SRID associated with the first geometry in the database, and assigns that to all geometries.

### `chunksize : int, default None`

If specified, return an iterator where chunksize is the number of rows to include in each chunk.

**See the documentation for `pandas.read_sql` for further explanation**

**of the following parameters:**

`index_col, coerce_float, parse_dates, params, chunksize`

## Returns:

`GeoDataFrame`

## Examples

PostGIS

```
>>> from sqlalchemy import create_engine
>>> db_connection_url = "postgresql://myusername:mypassword@myhost:5432/mydatabase"
>>> con = create_engine(db_connection_url)
>>> sql = "SELECT geom, highway FROM roads"
>>> df = geopandas.read_postgis(sql, con)
```

## SpatiaLite

```
>>> sql = "SELECT ST_AsBinary(geom) AS geom, highway FROM roads"
```

>>>

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx](#) 7.3.7.

# geopandas.read\_feather

`geopandas.read_feather(path, columns=None, **kwargs)`

[\[source\]](#)

Load a Feather object from the file path, returning a GeoDataFrame.

You can read a subset of columns in the file using the `columns` parameter. However, the structure of the returned GeoDataFrame will depend on which columns you read:

- if no geometry columns are read, this will raise a `ValueError` - you should use the pandas `read_feather` method instead.
- if the primary geometry column saved to this file is not included in `columns`, the first available geometry column will be set as the geometry column of the returned GeoDataFrame.

Supports versions 0.1.0, 0.4.0 and 1.0.0 of the GeoParquet specification at:

 [opengeospatial/geoparquet](#)

If ‘crs’ key is not present in the Feather metadata associated with the Parquet object, it will default to “OGC:CRS84” according to the specification.

Requires ‘pyarrow’ >= 0.17.

 **Added in version 0.8.**

## Parameters:

`path : str, path object`

`columns : list-like of strings, default=None`

If not None, only these columns will be read from the file. If the primary geometry column is not included, the first secondary geometry read from the file will be set as the geometry column of the returned GeoDataFrame. If no geometry columns are present, a `ValueError` will be raised.

`**kwargs`

Any additional kwargs passed to `pyarrow.feather.read_table()`.

## Returns:

`GeoDataFrame`

## Examples

```
>>> df = geopandas.read_feather("data.feather")
```

>>>

## Specifying columns to read:

```
>>> df = geopandas.read_feather(
... "data.feather",
... columns=["geometry", "pop_est"]
...)
```

>>>

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.read\_parquet

`geopandas.read_parquet(path, columns=None, storage_options=None, bbox=None, **kwargs)`

[\[source\]](#)

Load a Parquet object from the file path, returning a GeoDataFrame.

You can read a subset of columns in the file using the `columns` parameter. However, the structure of the returned GeoDataFrame will depend on which columns you read:

- if no geometry columns are read, this will raise a `ValueError` - you should use the pandas `read_parquet` method instead.
- if the primary geometry column saved to this file is not included in `columns`, the first available geometry column will be set as the geometry column of the returned GeoDataFrame.

Supports versions 0.1.0, 0.4.0 and 1.0.0 of the GeoParquet specification at:

 [opengeospatial/geoparquet](#)

If ‘crs’ key is not present in the GeoParquet metadata associated with the Parquet object, it will default to “OGC:CRS84” according to the specification.

Requires ‘pyarrow’.

 **Added in version 0.8.**

## Parameters:

`path : str, path object`

`columns : list-like of strings, default=None`

If not `None`, only these columns will be read from the file. If the primary geometry column is not included, the first secondary geometry read from the file will be set as the geometry column of the returned GeoDataFrame. If no geometry columns are present, a `ValueError` will be raised.

`storage_options : dict, optional`

Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with “`s3://`”, and “`gcs://`”) the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

When no storage options are provided and a filesystem is implemented by both

`pyarrow.fs` and `fsspec` (e.g. “`s3://`”) then the `pyarrow.fs` filesystem is preferred.

Provide the instantiated `fsspec` filesystem using the `filesystem` keyword if you wish to use its implementation.

## **bbox** : tuple, optional

Bounding box to be used to filter selection from geoparquet data. This is only usable if the data was saved with the bbox covering metadata. Input is of the tuple format (xmin, ymin, xmax, ymax).

## **\*\*kwargs**

Any additional kwargs passed to [`pyarrow.parquet.read\_table\(\)`](#).

## Returns:

**GeoDataFrame**

## Examples

```
>>> df = geopandas.read_parquet("data.parquet")>>>
```

Specifying columns to read:

```
>>> df = geopandas.read_parquet(
... "data.parquet",
... columns=["geometry", "pop_est"]
...)>>>
```



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.sjoin\_nearest

```
geopandas.sjoin_nearest(left_df, right_df, how='inner', max_distance=None,
lsuffix='left', rsuffix='right', distance_col=None, exclusive=False) [source]
```

Spatial join of two GeoDataFrames based on the distance between their geometries.

Results will include multiple output records for a single input record where there are multiple equidistant nearest or intersected neighbors.

Distance is calculated in CRS units and can be returned using the *distance\_col* parameter.

See the User Guide page

[https://geopandas.readthedocs.io/en/latest/docs/user\\_guide/mergingdata.html](https://geopandas.readthedocs.io/en/latest/docs/user_guide/mergingdata.html) for more details.

## Parameters:

**left\_df, right\_df : GeoDataFrames**

**how : string, default 'inner'**

The type of join:

- ‘left’: use keys from left\_df; retain only left\_df geometry column
- ‘right’: use keys from right\_df; retain only right\_df geometry column
- ‘inner’: use intersection of keys from both dfs; retain only left\_df geometry column

**max\_distance : float, default None**

Maximum distance within which to query for nearest geometry. Must be greater than 0. The max\_distance used to search for nearest items in the tree may have a significant impact on performance by reducing the number of input geometries that are evaluated for nearest items in the tree.

**lsuffix : string, default 'left'**

Suffix to apply to overlapping column names (left GeoDataFrame).

**rsuffix : string, default 'right'**

Suffix to apply to overlapping column names (right GeoDataFrame).

**distance\_col : string, default None**

If set, save the distances computed between matching geometries under a column of this name in the joined GeoDataFrame.

**exclusive : bool, default False**

If True, the nearest geometries that are equal to the input geometry will not be returned, default False.

## See also

### [sjoin](#)

binary predicate joins

### [GeoDataFrame.sjoin\\_nearest](#)

equivalent method

## Notes

Since this join relies on distances, results will be inaccurate if your geometries are in a geographic CRS.

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> import geodatasets
>>> groceries = geopandas.read_file(
... geodatasets.get_path("geoda.groceries")
...)
>>> chicago = geopandas.read_file(
... geodatasets.get_path("geoda.chicago_health")
...).to_crs(groceries.crs)
```

```
>>> chicago.head()
ComAreaID ... geometry
0 35 ... POLYGON ((-87.60914 41.84469, -87.60915 41.844...
1 36 ... POLYGON ((-87.59215 41.81693, -87.59231 41.816...
2 37 ... POLYGON ((-87.62880 41.80189, -87.62879 41.801...
3 38 ... POLYGON ((-87.60671 41.81681, -87.60670 41.816...
4 39 ... POLYGON ((-87.59215 41.81693, -87.59215 41.816...
[5 rows x 87 columns]
```

```
>>> groceries.head()
OBJECTID Ycoord ... Category geometry
0 16 41.973266 ... NaN MULTIPOLY ((-87.65661 41.97321))
1 18 41.696367 ... NaN MULTIPOLY ((-87.68136 41.69713))
2 22 41.868634 ... NaN MULTIPOLY ((-87.63918 41.86847))
3 23 41.877590 ... new MULTIPOLY ((-87.65495 41.87783))
4 27 41.737696 ... NaN MULTIPOLY ((-87.62715 41.73623))
[5 rows x 8 columns]
```

```
>>> groceries_w_communities = geopandas.sjoin_nearest(groceries, chicago)
>>> groceries_w_communities[["Chain", "community", "geometry"]].head(2)
 Chain community geometry
0 VIET HOA PLAZA UPTOWN MULTIPOLY ((1168268.672 1933554.35))
1 COUNTY FAIR FOODS MORGAN PARK MULTIPOLY ((1162302.618 1832900.224))
```

To include the distances:

```
>>> groceries_w_communities = geopandas.sjoin_nearest(groceries, chicago, distance_col="distances")
>>> groceries_w_communities[["Chain", "community", "distances"]].head(2)
 Chain community distances
0 VIET HOA PLAZA UPTOWN 0.0
1 COUNTY FAIR FOODS MORGAN PARK 0.0
```

In the following example, we get multiple groceries for Uptown because all results are equidistant (in this case zero because they intersect). In fact, we get 4 results in total:

```
>>> chicago_w_groceries = geopandas.sjoin_nearest(groceries, chicago, distance_col="distances")
>>> uptown_results = chicago_w_groceries[chicago_w_groceries["community"] == "UPTOWN"]
>>> uptown_results[["Chain", "community"]]
 Chain community
30 VIET HOA PLAZA UPTOWN
30 JEWEL OSCO UPTOWN
30 TARGET UPTOWN
30 Mariano's UPTOWN
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx 7.3.7](#).

# geopandas.overlay

```
geopandas.overlay(df1, df2, how='intersection', keep_geom_type=None,
make_valid=True)
```

[\[source\]](#)

Perform spatial overlay between two GeoDataFrames.

Currently only supports data GeoDataFrames with uniform geometry types, i.e. containing only (Multi)Polygons, or only (Multi)Points, or a combination of (Multi)LineString and LinearRing shapes. Implements several methods that are all effectively subsets of the union.

See the User Guide page [Set operations with overlay](#) for details.

## Parameters:

**df1 : GeoDataFrame**

**df2 : GeoDataFrame**

**how : string**

Method of spatial overlay: ‘intersection’, ‘union’, ‘identity’, ‘symmetric\_difference’ or ‘difference’.

**keep\_geom\_type : bool**

If True, return only geometries of the same geometry type as df1 has, if False, return all resulting geometries. Default is None, which will set keep\_geom\_type to True but warn upon dropping geometries.

**make\_valid : bool, default True**

If True, any invalid input geometries are corrected with a call to make\_valid(), if False, a ValueError is raised if any input geometries are invalid.

## Returns:

**df : GeoDataFrame**

GeoDataFrame with new set of polygons and attributes resulting from the overlay

## See also

[sjoin](#)

spatial join

[GeoDataFrame.overlay](#)

equivalent method

## Notes

Every operation in GeoPandas is planar, i.e. the potential third dimension is not taken into account.

## Examples

```
>>> from shapely.geometry import Polygon
>>> polys1 = geopandas.GeoSeries([Polygon([(0,0), (2,0), (2,2), (0,2)]),
... Polygon([(2,2), (4,2), (4,4), (2,4)])])
>>> polys2 = geopandas.GeoSeries([Polygon([(1,1), (3,1), (3,3), (1,3)]),
... Polygon([(3,3), (5,3), (5,5), (3,5)])])
>>> df1 = geopandas.GeoDataFrame({'geometry': polys1, 'df1_data':[1,2]})
>>> df2 = geopandas.GeoDataFrame({'geometry': polys2, 'df2_data':[1,2]})
```

```
>>> geopandas.overlay(df1, df2, how='union')
 df1_data df2_data geometry
0 1.0 1.0 POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))
1 2.0 1.0 POLYGON ((2 2, 2 3, 3 3, 3 2, 2 2))
2 2.0 2.0 POLYGON ((4 4, 4 3, 3 3, 3 4, 4 4))
3 1.0 NaN POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))
4 2.0 NaN MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4...
5 NaN 1.0 MULTIPOLYGON (((2 3, 2 2, 1 2, 1 3, 2 3)), ((3...
6 NaN 2.0 POLYGON ((3 5, 5 5, 5 3, 4 3, 4 4, 3 4, 3 5))
```

```
>>> geopandas.overlay(df1, df2, how='intersection')
 df1_data df2_data geometry
0 1.0 1 POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))
1 2.0 1 POLYGON ((2 2, 2 3, 3 3, 3 2, 2 2))
2 2.0 2 POLYGON ((4 4, 4 3, 3 3, 3 4, 4 4))
```

```
>>> geopandas.overlay(df1, df2, how='symmetric_difference')
 df1_data df2_data geometry
0 1.0 NaN POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))
1 2.0 NaN MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4...
2 NaN 1.0 MULTIPOLYGON (((2 3, 2 2, 1 2, 1 3, 2 3)), ((3...
3 NaN 2.0 POLYGON ((3 5, 5 5, 5 3, 4 3, 4 4, 3 4, 3 5))
```

```
>>> geopandas.overlay(df1, df2, how='difference')
 geometry df1_data
0 POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0)) 1
1 MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4... 2
```

```
>>> geopandas.overlay(df1, df2, how='identity')
 df1_data df2_data geometry
0 1.0 1.0 POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))
1 2.0 1.0 POLYGON ((2 2, 2 3, 3 3, 3 2, 2 2))
2 2.0 2.0 POLYGON ((4 4, 4 3, 3 3, 3 4, 4 4))
3 1.0 NaN POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))
4 2.0 NaN MULTIPOLYGON (((3 4, 3 3, 2 3, 2 4, 3 4)), ((4...)
```



# geopandas.clip

`geopandas.clip(gdf, mask, keep_geom_type=False, sort=False)`

[\[source\]](#)

Clip points, lines, or polygon geometries to the mask extent.

Both layers must be in the same Coordinate Reference System (CRS). The `gdf` will be clipped to the full extent of the clip object.

If there are multiple polygons in mask, data from `gdf` will be clipped to the total boundary of all polygons in mask.

If the `mask` is list-like with four elements `(minx, miny, maxx, maxy)`, a faster rectangle clipping algorithm will be used. Note that this can lead to slightly different results in edge cases, e.g. if a line would be reduced to a point, this point might not be returned. The geometry is clipped in a fast but possibly dirty way. The output is not guaranteed to be valid. No exceptions will be raised for topological errors.

## Parameters:

**gdf : GeoDataFrame or GeoSeries**

Vector layer (point, line, polygon) to be clipped to mask.

**mask : GeoDataFrame, GeoSeries, (Multi)Polygon, list-like**

Polygon vector layer used to clip `gdf`. The mask's geometry is dissolved into one geometric feature and intersected with `gdf`. If the mask is list-like with four elements `(minx, miny, maxx, maxy)`, `clip` will use a faster rectangle clipping (`clip_by_rect()`), possibly leading to slightly different results.

**keep\_geom\_type : boolean, default False**

If True, return only geometries of original type in case of intersection resulting in multiple geometry types or GeometryCollections. If False, return all resulting geometries (potentially mixed-types).

**sort : boolean, default False**

If True, the results will be sorted in ascending order using the geometries' indexes as the primary key.

## Returns:

**GeoDataFrame or GeoSeries**

Vector data (points, lines, polygons) from `gdf` clipped to polygon boundary from mask.

[GeoDataFrame.clip](#)

equivalent GeoDataFrame method

[GeoSeries.clip](#)

equivalent GeoSeries method

## Examples

Clip points (grocery stores) with polygons (the Near West Side community):

```
>>> import geopandas
>>> chicago = geopandas.read_file(
... geopandas.datasets.get_path("geoda.chicago_health")
...)
>>> near_west_side = chicago[chicago["community"] == "NEAR WEST SIDE"]
>>> groceries = geopandas.read_file(
... geopandas.datasets.get_path("geoda.groceries")
...).to_crs(chicago.crs)
>>> groceries.shape
(148, 8)
```

```
>>> nws_groceries = geopandas.clip(groceries, near_west_side)
>>> nws_groceries.shape
```

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# geopandas.tools.geocode

`geopandas.tools.geocode(strings, provider=None, **kwargs)`

[\[source\]](#)

Geocode a set of strings and get a GeoDataFrame of the resulting points.

## Parameters:

`strings : list or Series of addresses to geocode`

`provider : str or geopy.geocoder`

Specifies geocoding service to use. If none is provided, will use ‘photon’ (see the Photon’s terms of service at: <https://photon.komoot.io>).

Either the string name used by geopy (as specified in `geopy.geocoders.SERVICE_TO_GEOCODER`) or a geopy Geocoder instance (e.g., `geopy.geocoders.Photon`) may be used.

Some providers require additional arguments such as access keys See each geocoder’s specific parameters in `geopy.geocoders`

## Notes

Ensure proper use of the results by consulting the Terms of Service for your provider.

Geocoding requires geopy. Install it using ‘`pip install geopy`’. See also [!\[\]\(1ad1d2252f3c82a5c1c9ca318e77753a\_img.jpg\) geopy/geopy](#)

## Examples

```
>>> df = geopandas.tools.geocode(
... ["boston, ma", "1600 pennsylvania ave. washington, dc"]
...)
>>> df
 geometry address
0 POINT (-71.05863 42.35899) Boston, MA, United States
1 POINT (-77.03651 38.89766) 1600 Pennsylvania Ave NW, Washington, DC 20006...
```

# geopandas.tools.reverse\_geocode

`geopandas.tools.reverse_geocode(points, provider=None, **kwargs)`

[\[source\]](#)

Reverse geocode a set of points and get a GeoDataFrame of the resulting addresses.

The points

## Parameters:

**points : list or Series of Shapely Point objects.**

x coordinate is longitude y coordinate is latitude

**provider : str or geopy.geocoder (opt)**

Specifies geocoding service to use. If none is provided, will use ‘photon’ (see the Photon’s terms of service at: <https://photon.komoot.io>).

Either the string name used by geopy (as specified in

geopy.geocoders.SERVICE\_TO\_GEOCODER) or a geopy Geocoder instance (e.g., geopy.geocoders.Photon) may be used.

Some providers require additional arguments such as access keys See each geocoder’s specific parameters in geopy.geocoders

## Notes

Ensure proper use of the results by consulting the Terms of Service for your provider.

Reverse geocoding requires geopy. Install it using ‘pip install geopy’. See also [geopy/geopy](#)

## Examples

```
>>> from shapely.geometry import Point
>>> df = geopandas.tools.reverse_geocode(
... [Point(-71.0594869, 42.3584697), Point(-77.0365305, 38.8977332)])
...)
>>> df
 geometry address
0 POINT (-71.05941 42.35837) 29 Court Sq, Boston, MA 02108, United States
1 POINT (-77.03641 38.89766) 1600 Pennsylvania Ave NW, Washington, DC 20006...
```

# geopandas.tools.collect

`geopandas.tools.collect(x, multi=False)`

[\[source\]](#)

Collect single part geometries into their Multi\* counterpart

## Parameters:

**x** : *an iterable or Series of Shapely geometries, a GeoSeries, or*

a single Shapely geometry

**multi** : *boolean, default False*

if True, force returned geometries to be Multi\* even if they only have one component.

# geopandas.points\_from\_xy

`geopandas.points_from_xy(x, y, z=None, crs=None)`

[\[source\]](#)

Generate GeometryArray of shapely Point geometries from x, y(, z) coordinates.

In case of geographic coordinates, it is assumed that longitude is captured by `x` coordinates and latitude by `y`.

## Parameters:

`x, y, z : iterable`

`crs : value, optional`

Coordinate Reference System of the geometry objects. Can be anything accepted by

`pyproj.CRS.from_user_input()`, such as an authority string (eg “EPSG:4326”) or a WKT string.

## Returns:

`output : GeometryArray`

## Examples

```
>>> import pandas as pd
>>> df = pd.DataFrame({'x': [0, 1, 2], 'y': [0, 1, 2], 'z': [0, 1, 2]})
>>> df
 x y z
0 0 0 0
1 1 1 1
2 2 2 2
>>> geometry = geopandas.points_from_xy(x=[1, 0], y=[0, 1])
>>> geometry = geopandas.points_from_xy(df['x'], df['y'], df['z'])
>>> gdf = geopandas.GeoDataFrame(
... df, geometry=geopandas.points_from_xy(df['x'], df['y']))
```

Having geographic coordinates:

```
>>> df = pd.DataFrame({'longitude': [-140, 0, 123], 'latitude': [-65, 1, 48]})
>>> df
 longitude latitude
0 -140 -65
1 0 1
2 123 48
>>> geometry = geopandas.points_from_xy(df.longitude, df.latitude, crs="EPSG:4326")
```



# Spatial index

GeoPandas will use the STRtree implementation provided by the Shapely ([shapely.STRtree](#))

`GeoSeries.sindex` creates a spatial index, which can use the methods and properties documented below.

## Constructor

`GeoSeries.sindex`

Generate the spatial index

## Spatial index object

The spatial index object returned from `GeoSeries.sindex` has the following methods:

<code>intersection</code> (coordinates)	Compatibility wrapper for rtree.index.Index.intersection, use <code>query</code> instead.
<code>is_empty</code>	Check if the spatial index is empty
<code>nearest</code> (geometry[, return_all, ...])	Return the nearest geometry in the tree for each input geometry in <code>geometry</code> .
<code>query</code> (geometry[, predicate, sort, distance, ...])	Return the integer indices of all combinations of each input geometry and tree geometries where the bounding box of each input geometry intersects the bounding box of a tree geometry.
<code>size</code>	Size of the spatial index
<code>valid_query_predicates</code>	Returns valid predicates for the spatial index.

The spatial index offers the full capability of `shapely.STRtree`.



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# geopandas.sindex.SpatialIndex.is\_empty

*property* `SpatialIndex.is_empty`

[\[source\]](#)

Check if the spatial index is empty

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries(geopandas.points_from_xy(range(10), range(10)))
>>> s
0 POINT (0 0)
1 POINT (1 1)
2 POINT (2 2)
3 POINT (3 3)
4 POINT (4 4)
5 POINT (5 5)
6 POINT (6 6)
7 POINT (7 7)
8 POINT (8 8)
9 POINT (9 9)
dtype: geometry
```

```
>>> s.sindex.is_empty
False
```

```
>>> s2 = geopandas.GeoSeries()
>>> s2.sindex.is_empty
True
```

# geopandas.sindex.SpatialIndex.nearest

`SpatialIndex.nearest(geometry, return_all=True, max_distance=None, return_distance=False, exclusive=False)`

[\[source\]](#)

Return the nearest geometry in the tree for each input geometry in `geometry`.

If multiple tree geometries have the same distance from an input geometry, multiple results will be returned for that input geometry by default. Specify `return_all=False` to only get a single nearest geometry (non-deterministic which nearest is returned).

In the context of a spatial join, input geometries are the “left” geometries that determine the order of the results, and tree geometries are “right” geometries that are joined against the left geometries. If `max_distance` is not set, this will effectively be a left join because every geometry in `geometry` will have a nearest geometry in the tree. However, if `max_distance` is used, this becomes an inner join, since some geometries in `geometry` may not have a match in the tree.

For performance reasons, it is highly recommended that you set the `max_distance` parameter.

## Parameters:

`geometry : {shapely.geometry, GeoSeries, GeometryArray, numpy.array of Shapely geometries}`

A single shapely geometry, one of the GeoPandas geometry iterables (GeoSeries, GeometryArray), or a numpy array of Shapely geometries to query against the spatial index.

`return_all : bool, default True`

If there are multiple equidistant or intersecting nearest geometries, return all those geometries instead of a single nearest geometry.

`max_distance : float, optional`

Maximum distance within which to query for nearest items in tree. Must be greater than 0. By default None, indicating no distance limit.

`return_distance : bool, optional`

If True, will return distances in addition to indexes. By default False

`exclusive : bool, optional`

if True, the nearest geometries that are equal to the input geometry will not be returned. By default False. Requires Shapely >= 2.0.

## Returns:

`Indices or tuple of (indices, distances)`

Indices is an ndarray of shape (2,n) and distances (if present) an ndarray of shape (n). The first subarray of indices contains input geometry indices. The second subarray of indices contains tree geometry indices.

## Examples

```
>>> from shapely.geometry import Point, box
>>> s = geopandas.GeoSeries(geopandas.points_from_xy(range(10), range(10)))
>>> s.head()
0 POINT (0 0)
1 POINT (1 1)
2 POINT (2 2)
3 POINT (3 3)
4 POINT (4 4)
dtype: geometry
```

```
>>> s.sindex.nearest(Point(1, 1))
array([[0],
 [1]])
```

```
>>> s.sindex.nearest([box(4.9, 4.9, 5.1, 5.1)])
array([[0],
 [5]])
```

```
>>> s2 = geopandas.GeoSeries(geopandas.points_from_xy([7.6, 10], [7.6, 10]))
>>> s2
0 POINT (7.6 7.6)
1 POINT (10 10)
dtype: geometry
```

```
>>> s.sindex.nearest(s2)
array([[0, 1],
 [8, 9]])
```

# geopandas.sindex.SpatialIndex.query

`SpatialIndex.query(geometry, predicate=None, sort=False, distance=None, output_format='tuple')`

[\[source\]](#)

Return the integer indices of all combinations of each input geometry and tree geometries where the bounding box of each input geometry intersects the bounding box of a tree geometry.

If the input geometry is a scalar, this returns an array of shape (n, ) with the indices of the matching tree geometries. If the input geometry is an array\_like, this returns an array with shape (2,n) where the subarrays correspond to the indices of the input geometries and indices of the tree geometries associated with each. To generate an array of pairs of input geometry index and tree geometry index, simply transpose the result.

If a predicate is provided, the tree geometries are first queried based on the bounding box of the input geometry and then are further filtered to those that meet the predicate when comparing the input geometry to the tree geometry: `predicate(geometry, tree_geometry)`.

The ‘dwithin’ predicate requires GEOS >= 3.10.

Bounding boxes are limited to two dimensions and are axis-aligned (equivalent to the `bounds` property of a geometry); any Z values present in input geometries are ignored when querying the tree.

Any input geometry that is None or empty will never match geometries in the tree.

## Parameters:

`geometry : shapely.Geometry or array-like of geometries (numpy.ndarray, GeoSeries, GeometryArray)`

A single shapely geometry or array of geometries to query against the spatial index. For array-like, accepts both GeoPandas geometry iterables (GeoSeries, GeometryArray) or a numpy array of Shapely geometries.

`predicate : {None, “contains”, “contains_properly”, “covered_by”, “covers”, “crosses”, “intersects”, “overlaps”, “touches”, “within”, “dwithin”}, optional`

If predicate is provided, the input geometries are tested using the predicate function against each item in the tree whose extent intersects the envelope of the input geometry:

`predicate(input_geometry, tree_geometry)`. If possible, prepared geometries are used to help speed up the predicate operation.

`sort : bool, default False`

If True, the results will be sorted in ascending order. In case of 2D array, the result is sorted lexicographically using the geometries’ indexes as the primary key and the sindex’s indexes as the secondary key. If False, no additional sorting is applied (results are often sorted but there is no guarantee).

## distance : number or array\_like, optional

Distances around each input geometry within which to query the tree for the ‘dwithin’ predicate. If array\_like, shape must be broadcastable to shape of geometry. Required if `predicate='dwithin'`.

### Returns:

#### ndarray with shape (n,) if geometry is a scalar

Integer indices for matching geometries from the spatial index tree geometries.

OR

#### ndarray with shape (2, n) if geometry is an array\_like

The first subarray contains input geometry integer indices. The second subarray contains tree geometry integer indices.

### Notes

In the context of a spatial join, input geometries are the “left” geometries that determine the order of the results, and tree geometries are “right” geometries that are joined against the left geometries. This effectively performs an inner join, where only those combinations of geometries that can be joined based on overlapping bounding boxes or optional predicate are returned.

### Examples

```
>>> from shapely.geometry import Point, box
>>> s = geopandas.GeoSeries(geopandas.points_from_xy(range(10), range(10)))
>>> s
0 POINT (0 0)
1 POINT (1 1)
2 POINT (2 2)
3 POINT (3 3)
4 POINT (4 4)
5 POINT (5 5)
6 POINT (6 6)
7 POINT (7 7)
8 POINT (8 8)
9 POINT (9 9)
dtype: geometry
```

Querying the tree with a scalar geometry:

```
>>> s.sindex.query(box(1, 1, 3, 3))
array([1, 2, 3])
```

```
>>> s.sindex.query(box(1, 1, 3, 3), predicate="contains")
array([2])
```

## Querying the tree with an array of geometries:

```
>>> s2 = geopandas.GeoSeries([box(2, 2, 4, 4), box(5, 5, 6, 6)])
>>> s2
0 POLYGON ((4 2, 4 4, 2 4, 2 2, 4 2))
1 POLYGON ((6 5, 6 6, 5 6, 5 5, 6 5))
dtype: geometry
```

```
>>> s.sindex.query(s2)
array([[0, 0, 0, 1, 1],
 [2, 3, 4, 5, 6]])
```

```
>>> s.sindex.query(s2, predicate="contains")
array([[0],
 [3]])
```

```
>>> s.sindex.query(box(1, 1, 3, 3), predicate="dwithin", distance=0)
array([1, 2, 3])
```

```
>>> s.sindex.query(box(1, 1, 3, 3), predicate="dwithin", distance=2)
array([0, 1, 2, 3, 4])
```

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# geopandas.sindex.SpatialIndex.size

*property* **SpatialIndex.size**

[\[source\]](#)

Size of the spatial index

Number of leaves (input geometries) in the index.

## Examples

```
>>> from shapely.geometry import Point
>>> s = geopandas.GeoSeries(geopandas.points_from_xy(range(10), range(10)))
>>> s
0 POINT (0 0)
1 POINT (1 1)
2 POINT (2 2)
3 POINT (3 3)
4 POINT (4 4)
5 POINT (5 5)
6 POINT (6 6)
7 POINT (7 7)
8 POINT (8 8)
9 POINT (9 9)
dtype: geometry
```

```
>>> s.sindex.size
10
```



Verifying you are human. This may take a few seconds.

geopandas.org needs to review the security of your connection before proceeding.

# Testing

GeoPandas includes specific functions to test its objects.

`testing.assert_geoseries_equal`(left, right)

Test util for checking that two GeoSeries are equal.

`testing.assert_geodataframe_equal`(left, right)

Check that two GeoDataFrames are equal/

# geopandas.testing.assert\_geoseries\_equal

```
geopandas.testing.assert_geoseries_equal(left, right, check_dtype=True,
check_index_type=False, check_series_type=True, check_less_precise=False,
check_geom_type=False, check_crs=True, normalize=False)
```

[\[source\]](#)

Test util for checking that two GeoSeries are equal.

## Parameters:

**left, right : two GeoSeries**

**check\_dtype : bool, default False**

If True, check geo dtype [only included so it's a drop-in replacement for assert\_series\_equal].

**check\_index\_type : bool, default False**

Check that index types are equal.

**check\_series\_type : bool, default True**

Check that both are same type (*and* are GeoSeries). If False, will attempt to convert both into GeoSeries.

**check\_less\_precise : bool, default False**

If True, use geom\_equals\_exact with relative error of 0.5e-6. If False, use geom\_equals.

**check\_geom\_type : bool, default False**

If True, check that all the geom types are equal.

**check\_crs: bool, default True**

If *check\_series\_type* is True, then also check that the crs matches.

**normalize: bool, default False**

If True, normalize the geometries before comparing equality. Typically useful with

`check_less_precise=True`, which uses `geom_equals_exact` and requires exact coordinate order.

# geopandas.testing.assert\_geodataframe\_equal

```
geopandas.testing.assert_geodataframe_equal(left, right, check_dtype=True,
check_index_type='equiv', check_column_type='equiv', check_frame_type=True,
check_like=False, check_less_precise=False, check_geom_type=False,
check_crs=True, normalize=False)
```

[\[source\]](#)

Check that two GeoDataFrames are equal/

## Parameters:

**left, right : two GeoDataFrames**

**check\_dtype : bool, default True**

Whether to check the DataFrame dtype is identical.

**check\_index\_type, check\_column\_type : bool, default 'equiv'**

Check that index types are equal.

**check\_frame\_type : bool, default True**

Check that both are same type (and are GeoDataFrames). If False, will attempt to convert both into GeoDataFrame.

**check\_like : bool, default False**

If true, ignore the order of rows & columns

**check\_less\_precise : bool, default False**

If True, use geom\_equals\_exact. if False, use geom\_equals.

**check\_geom\_type : bool, default False**

If True, check that all the geom types are equal.

**check\_crs: bool, default True**

If `check_frame_type` is True, then also check that the crs matches.

**normalize: bool, default False**

If True, normalize the geometries before comparing equality. Typically useful with

`check_less_precise=True`, which uses `geom_equals_exact` and requires exact coordinate order.

# Changelog

## Version 1.0.1

Bug fixes:

- Support a named datetime or object dtype index in `explore()` (#3360, #3364).
- Fix a regression preventing a Series as an argument for geometric methods (#3363)

## Version 1.0.0 (June 24, 2024)

Notes on dependencies:

- GeoPandas 1.0 drops support for shapely<2 and PyGEOS. The only geometry engine that is currently supported is shapely >= 2. As a consequence, spatial indexing based on the rtree package has also been removed (#3035).
- The I/O engine now defaults to Pyogrio which is now installed with GeoPandas instead of Fiona (#3223).

New methods:

- Added `count_geometries` method from shapely to GeoSeries/GeoDataframe (#3154).
- Added `count_interior_rings` method from shapely to GeoSeries/GeoDataframe (#3154)
- Added `relate_pattern` method from shapely to GeoSeries/GeoDataframe (#3211).
- Added `intersection_all` method from shapely to GeoSeries/GeoDataframe (#3228).
- Added `line_merge` method from shapely to GeoSeries/GeoDataframe (#3214).
- Added `set_precision` and `get_precision` methods from shapely to GeoSeries/GeoDataframe (#3175).
- Added `count_coordinates` method from shapely to GeoSeries/GeoDataframe (#3026).
- Added `minimum_clearance` method from shapely to GeoSeries/GeoDataframe (#2989).
- Added `shared_paths` method from shapely to GeoSeries/GeoDataframe (#3215).
- Added `is_ccw` method from shapely to GeoSeries/GeoDataframe (#3027).
- Added `is_closed` attribute from shapely to GeoSeries/GeoDataframe (#3092).
- Added `force_2d` and `force_3d` methods from shapely to GeoSeries/GeoDataframe (#3090).
- Added `voronoi_polygons` method from shapely to GeoSeries/GeoDataframe (#3177).
- Added `contains_properly` method from shapely to GeoSeries/GeoDataframe (#3105).
- Added `build_area` method exposing `build_area` shapely to GeoSeries/GeoDataframe (#3202).
- Added `snap` method from shapely to GeoSeries/GeoDataframe (#3086).

- Added `transform` method from shapely to GeoSeries/GeoDataFrame (#3075).
- Added `get_geometry` method from shapely to GeoSeries/GeoDataFrame (#3287).
- Added `dwithin` method to check for a “distance within” predicate on GeoSeries/GeoDataFrame (#3153).
- Added `to_geo_dict` method to generate GeoJSON-like dictionary from a GeoDataFrame (#3132).
- Added `polygonize` method exposing both `polygonize` and `polygonize_full` from shapely to GeoSeries/GeoDataFrame (#2963).
- Added `is_valid_reason` method from shapely to GeoSeries/GeoDataFrame (#3176).
- Added `to_arrow` method and `from_arrow` class method to GeoSeries/GeoDataFrame to export and import to/from Arrow data with GeoArrow extension types (#3219, #3301).

New features and improvements:

- Added `predicate="dwithin"` option and `distance` argument to the `sindex.query()` method and `sjoin` (#2882).
- GeoSeries and GeoDataFrame `__repr__` now trims trailing zeros for a more readable output (#3087).
- Add `on_invalid` parameter to `from_wkt` and `from_wkb` (#3110).
- `make_valid` option in `overlay` now uses the `make_valid` method instead of `buffer(0)` (#3113).
- Passing `"geometry"` as `dtype` to `pd.read_csv` will now return a GeoSeries for the specified columns (#3101).
- Added support to `read_file` for the `mask` keyword for the pyogrio engine (#3062).
- Added support to `read_file` for the `columns` keyword for the fiona engine (#3133).
- Added support to `to_parquet` and `read_parquet` for writing and reading files using the GeoArrow-based native geometry encoding of GeoParquet 1.1 (#3253, #3275).
- Add `sort` keyword to `clip` method for GeoSeries and GeoDataFrame to allow optional preservation of the original order of observations (#3233).
- Added `show_bbox`, `drop_id` and `to_wgs84` arguments to allow further customization of `GeoSeries.to_json` (#3226).
- `explore` now supports `GeoDataFrame`s with additional columns containing datetimes, uuids and other non JSON serializable objects (#3261).
- The `GeoSeries.fillna` method now supports the `limit` keyword (#3290).
- Added `on_attribute` option argument to the `sjoin()` method, allowing to restrict joins to the observations with matching attributes. (#3231)
- Added support for `bbox` covering encoding in geoparquet. Can filter reading of parquet files based on a bounding box, and write out a bounding box column to parquet files (#3282).
- `align` keyword in binary methods now defaults to `None`, treated as True. Explicit True will silence the warning about mismatched indices (#3212).
- `GeoSeries.set_crs` can now be used to remove CRS information by passing `crs=None, allow_override=True` (#3316).

- Added `autolim` keyword argument to `GeoSeries.plot()` and `GeoDataFrame.plot()` (#2817).
- Added `metadata` parameter to `GeoDataFrame.to_file` (#2850)
- Updated documentation to clarify that passing a named (Geo)Series as the `geometry` argument to the GeoDataFrame constructor will not use the name but will always produce a GeoDataFrame with an active geometry column named “geometry” (#3337).
- `read_postgis` will query the `spatial_ref_sys` table to determine the CRS authority instead of its current behaviour of assuming EPSG. In the event the `spatial_ref_sys` table is not present, or the SRID is not present, `read_postgis` will fallback on assuming EPSG CRS authority. (#3329)

Backwards incompatible API changes:

- The `sjoin` method will now preserve the name of the index of the right GeoDataFrame, if it has one, instead of always using `"index_right"` as the name for the resulting column in the return value (#846, #2144).
- GeoPandas now raises a `ValueError` when an unaligned Series is passed as a method argument to avoid confusion of whether the automatic alignment happens or not (#3271).
- The deprecated default value of `GeoDataFrame/ GeoSeries explode(..., index_parts=True)` is now set to false for consistency with pandas (#3174).
- The behaviour of `set_geometry` has been changed when passed a (Geo)Series `ser` with a name. The new active geometry column name in this case will be `ser.name`, if not `None`, rather than the previous active geometry column name. This means that if the new and old names are different, then both columns will be preserved in the GeoDataFrame. To replicate the previous behaviour, you can instead call `gdf.set_geometry(ser.rename(gdf.active_geometry_name))` (#3237). Note that this behaviour change does not affect the `GeoDataframe` constructor, passing a named GeoSeries `ser` to `GeoDataFrame(df, geometry=ser)` will always produce a GeoDataFrame with a geometry column named “geometry” to preserve backwards compatibility. If you would like to instead propagate the name of `ser` when constructing a GeoDataFrame, you can instead call `df.set_geometry(ser)` or `GeoDataFrame(df, geometry=ser).rename_geometry(ser.name)` (#3337).
- `delaunay_triangles` now considers all geometries together when creating the Delaunay triangulation instead of performing the operation element-wise. If you want to generate Delaunay triangles for each geometry separately, use `shapely.delaunay_triangles` instead. (#3273)
- Reading a data source that does not have a geometry field using `read_file` now returns a Pandas DataFrame instead of a GeoDataFrame with an empty `geometry` column.

Enforced deprecations:

- The deprecation of `geopandas.datasets` has been enforced and the module has been removed. New sample datasets are now available in the `geodatasets` package (#3084).
- Many longstanding deprecated functions, methods and properties have been removed (#3174), (#3190)
  - Removed deprecated functions `geopandas.io.read_file`, `geopandas.io.to_file` and `geopandas.io.sql.read_postgis`. `geopandas.read_file`, `geopandas.read_postgis` and the

GeoDataFrame/GeoSeries `to_file(...)` method should be used instead.

- Removed deprecated `GeometryArray.data` property, `np.asarray(...)` or the `to_numpy()` method should be used instead.
- Removed deprecated `sindex.query_bulk` method, using `sindex.query` instead.
- Removed deprecated `sjoin` parameter `op`, `predicate` should be supplied instead.
- Removed deprecated GeoSeries/ GeoDataFrame methods `__xor__`, `__or__`, `__and__` and `__sub__`. Instead use methods `symmetric_difference`, `union`, `intersection` and `difference` respectively.
- Removed deprecated plotting functions `plot_polygon_collection`, `plot_linestring_collection` and `plot_point_collection`, use the GeoSeries/GeoDataFrame `.plot` method directly instead.
- Removed deprecated GeoSeries/GeoDataFrame `.plot` parameters `axes` and `colormap`, instead use `ax` and `cmap` respectively.
- Removed compatibility for specifying the `version` keyword in `to_parquet` and `to_feather`. This keyword will now be passed through to pyarrow and use `schema_version` to specify the GeoParquet specification version (#3334).

New deprecations:

- `unary_union` attribute is now deprecated and replaced by the `union_all()` method (#3007) allowing opting for a faster union algorithm for coverages (#3151).
- The `include_fields` and `ignore_fields` keywords in `read_file()` are deprecated for the default pyogrio engine. Currently those are translated to the `columns` keyword for backwards compatibility, but you should directly use the `columns` keyword instead to select which columns to read (#3133).
- The `drop` keyword in `set_geometry` has been deprecated, and in future the `drop=True` behaviour will be removed (#3237). To prepare for this change, you should remove any explicit `drop=False` calls in your code (the default behaviour already is the same as `drop=False`). To replicate the previous `drop=True` behaviour you should replace `gdf.set_geometry(new_geo_col, drop=True)` with

```
geo_col_name = gdf.active_geometry_name
gdf.set_geometry(new_geo_col).drop(columns=geo_col_name).rename_geometry(geo_col_name)
```

- The `geopandas.use_pygeos` option has been deprecated and will be removed in GeoPandas 1.1 (#3283)
- Manual overriding of an existing CRS of a GeoSeries or GeoDataFrame by setting the `crs` property has been deprecated and will be disabled in future. Use the `set_crs()` method instead (#3085).

Bug fixes:

- Fix `GeoDataFrame.merge()` incorrectly returning a `DataFrame` instead of a `GeoDataFrame` when the `suffixes` argument is applied to the active geometry column (#2933).

- Fix bug in `GeoDataFrame` constructor where if `geometry` is given a named `GeoSeries` the name was not used as the active geometry column name (#3237).
- Fix bug in `GeoSeries` constructor when passing a Series and specifying a `crs` to not change the original input data (#2492).
- Fix regression preventing reading from file paths containing hashes in `read_file` with the fiona engine (#3280). An analogous fix for pyogrio is included in pyogrio 0.8.1.
- Fix `to_parquet` to write correct metadata in case of 3D geometries (#2824).
- Fixes for compatibility with psycopg (#3167).
- Fix to allow appending dataframes with no CRS to PostGIS tables with no CRS (#3328)
- Fix plotting of all-empty GeoSeries using `explore` (#3316).

## Version 0.14.4 (April 26, 2024)

- Several fixes for compatibility with the upcoming pandas 3.0, numpy 2.0 and fiona 1.10 releases.

## Version 0.14.3 (Jan 31, 2024)

- Several fixes for compatibility with the latest pandas 2.2 release.
- Fix bug in `pandas.concat` CRS consistency checking where CRS differing by WKT whitespace only were treated as incompatible (#3023).

## Version 0.14.2 (Jan 4, 2024)

- Fix regression in `overlay` where using `buffer(0)` instead of `make_valid` internally produced invalid results (#3074).
- Fix `explore()` method when the active geometry contains missing and empty geometries (#3094).

## Version 0.14.1 (Nov 11, 2023)

- The Parquet and Feather IO functions now support the latest 1.0.0 version of the GeoParquet specification ([geoparquet.org](http://geoparquet.org)) (#2663).
- Fix `read_parquet` and `read_feather` for [CVE-2023-47248](#) (#3070).

## Version 0.14 (Sep 15, 2023)

GeoPandas will use Shapely 2.0 by default instead of PyGEOS when both Shapely  $\geq$  2.0 and PyGEOS are installed. PyGEOS will continue to be used by default when PyGEOS is installed alongside Shapely <

## API changes:

- `seed` keyword in `sample_points` is deprecated. Use `rng` instead. (#2913).

## New methods:

- Added `concave_hull` method from shapely to GeoSeries/GeoDataframe (#2903).
- Added `delaunay_triangles` method from shapely to GeoSeries/GeoDataframe (#2907).
- Added `extract_unique_points` method from shapely to GeoSeries/GeoDataframe (#2915).
- Added `frechet_distance()` method from shapely to GeoSeries/GeoDataframe (#2929).
- Added `hausdorff_distance` method from shapely to GeoSeries/GeoDataframe (#2909).
- Added `minimum_rotated_rectangle` method from shapely to GeoSeries/GeoDataframe (#2541).
- Added `offset_curve` method from shapely to GeoSeries/GeoDataframe (#2902).
- Added `remove_repeated_points` method from shapely to GeoSeries/GeoDataframe (#2940).
- Added `reverse` method from shapely to GeoSeries/GeoDataframe (#2988).
- Added `segmentize` method from shapely to GeoSeries/GeoDataFrame (#2910).
- Added `shortest_line` method from shapely to GeoSeries/GeoDataframe (#2960).

## New features and improvements:

- Added `exclusive` parameter to `sjoin_nearest` method for Shapely >= 2.0 (#2877)
- Added `GeoDataFrame.active_geometry_name` property returning the active geometry column's name or None if no active geometry column is set.
- The `to_file()` method will now automatically detect the FlatGeoBuf driver for files with the `.fgb` extension (#2958)

## Bug fixes:

- Fix ambiguous error when GeoDataFrame is initialized with a column called `"crs"` (#2944)
- Fix a color assignment in `explore` when using `UserDefined` bins (#2923)
- Fix bug in `apply` with `axis=1` where the given user defined function returns nested data in the geometry column (#2959)
- Properly infer schema for `np.int32` and `pd.Int32Dtype` columns (#2950)
- `assert_geodataframe_equal` now handles GeoDataFrames with no active geometry (#2498)

## Notes on (optional) dependencies:

- GeoPandas 0.14 drops support for Python 3.8 and pandas 1.3 and below (the minimum supported pandas version is now 1.4). Further, the minimum required versions for the listed dependencies have now changed to shapely 1.8.0, fiona 1.8.21, pyproj 3.3.0 and matplotlib 3.5.0 (#3001)

- `geom_almost_equals()` methods have been deprecated and `geom_equals_exact()` should be used instead (#2604).

## Version 0.13.2 (Jun 6, 2023)

Bug fix:

- Fix a regression in reading from local file URIs (`file://...`) using `geopandas.read_file` (#2948).

## Version 0.13.1 (Jun 5, 2023)

Bug fix:

- Fix a regression in reading from URLs using `geopandas.read_file` (#2908). This restores the behaviour to download all data up-front before passing it to the underlying engine (fiona or pyogrio), except if the server supports partial requests (to support reading a subset of a large file).

## Version 0.13 (May 6, 2023)

New methods:

- Added `sample_points` method to sample random points from Polygon or LineString geometries (#2860).
- New `hilbert_distance()` method that calculates the distance along a Hilbert curve for each geometry in a GeoSeries/GeoDataFrame (#2297).
- Support for sorting geometries (for example, using `sort_values()`) based on the distance along the Hilbert curve (#2070).
- Added `get_coordinates()` method from shapely to GeoSeries/GeoDataFrame (#2624).
- Added `minimum_bounding_circle()` method from shapely to GeoSeries/GeoDataFrame (#2621).
- Added `minimum_bounding_radius()` as GeoSeries method (#2827).

Other new features and improvements:

- The Parquet and Feather IO functions now support the latest 1.0.0-beta.1 version of the GeoParquet specification (<geoparquet.org>) (#2663).
- Added support to fill missing values in `GeoSeries.fillna` via another `GeoSeries` (#2535).
- Support specifying `min_zoom` and `max_zoom` inside the `map_kwds` argument for `.explore()` (#2599).

- Added support for append (`mode="a"` or `append=True`) in `to_file()` using `engine="pyogrio"` (#2788).
- Added a `to_wgs84` keyword to `to_json` allowing automatic re-projecting to follow the 2016 GeoJSON specification (#416).
- `to_json` output now includes a `"crs"` field if the CRS is not the default WGS84 (#1774).
- Improve error messages when accessing the `geometry` attribute of GeoDataFrame without an active geometry column related to the default name `"geometry"` being provided in the constructor (#2577)

Deprecations and compatibility notes:

- Added warning that `unary_union` will return `'GEOMETRYCOLLECTION EMPTY'` instead of None for all-None GeoSeries. (#2618)
- The `query_bulk()` method of the spatial index `.sindex` property is deprecated in favor of `query()` (#2823).

Bug fixes:

- Ensure that GeoDataFrame created from DataFrame is a copy, not a view (#2667)
- Fix mismatch between geometries and colors in `plot()` if an empty or missing geometry is present (#2224)
- Escape special characters to avoid TemplateSyntaxError in `explore()` (#2657)
- Fix `to_parquet`/`to_feather` to not write an invalid bbox (with NaNs) in the metadata in case of an empty GeoDataFrame (#2653)
- Fix `to_parquet`/`to_feather` to use correct WKB flavor for 3D geometries (#2654)
- Fix `read_file` to avoid reading all file bytes prior to calling Fiona or Pyogrio if provided a URL as input (#2796)
- Fix `copy()` downcasting GeoDataFrames without an active geometry column to a DataFrame (#2775)
- Fix geometry column name propagation when GeoDataFrame columns are a multiindex (#2088)
- Fix `iterfeatures()` method of GeoDataFrame to correctly handle non-scalar values when `na='drop'` is specified (#2811)
- Fix issue with passing custom legend labels to `plot` (#2886)

Notes on (optional) dependencies:

- GeoPandas 0.13 drops support pandas 1.0.5 (the minimum supported pandas version is now 1.1). Further, the minimum required versions for the listed dependencies have now changed to shapely 1.7.1, fiona 1.8.19, pyproj 3.0.1 and matplotlib 3.3.4 (#2655)

## Version 0.12.2 (December 10, 2022)

Bug fixes:

- Correctly handle geometries with Z dimension in `to_crs()` when using PyGEOS or Shapely >= 2.0 (previously the z coordinates were lost) (#1345).
- Assign Crimea to Ukraine in the `natural_earth_lowres` built-in dataset (#2670)

## Version 0.12.1 (October 29, 2022)

Small bug-fix release removing the shapely<2 pin in the installation requirements.

## Version 0.12 (October 24, 2022)

The highlight of this release is the support for Shapely 2.0. This makes it possible to test Shapely 2.0 (currently 2.0b1) alongside GeoPandas.

Note that if you also have PyGEOS installed, you need to set an environment variable (`USE_PYGEOS=0`) before importing geopandas to actually test Shapely 2.0 features instead of PyGEOS. See [https://geopandas.org/en/latest/getting\\_started/install.html#using-the-optional-pygeos-dependency](https://geopandas.org/en/latest/getting_started/install.html#using-the-optional-pygeos-dependency) for more details.

New features and improvements:

- Added `normalize()` method from shapely to GeoSeries/GeoDataFrame (#2537).
- Added `make_valid()` method from shapely to GeoSeries/GeoDataFrame (#2539).
- Added `where` filter to `read_file` (#2552).
- Updated the distributed natural earth datasets (`natural_earth_lowres` and `natural_earth_cities`) to version 5.1 (#2555).

Deprecations and compatibility notes:

- Accessing the `crs` of a `GeoDataFrame` without active geometry column was deprecated and this now raises an `AttributeError` (#2578).
- Resolved colormap-related warning in `.explore()` for recent Matplotlib versions (#2596).

Bug fixes:

- Fix cryptic error message in `geopandas.clip()` when clipping with an empty geometry (#2589).
- Accessing `gdf.geometry` where the active geometry column is missing, and a column named `"geometry"` is present will now raise an `AttributeError`, rather than returning `gdf["geometry"]` (#2575).
- Combining GeoSeries/GeoDataFrames with `pandas.concat` will no longer silently override CRS information if not all inputs have the same CRS (#2056).

# Version 0.11.1 (July 24, 2022)

Small bug-fix release:

- Fix regression (RecursionError) in reshape methods such as `unstack()` and `pivot()` involving MultiIndex, or GeoDataFrame construction with MultiIndex (#2486).
- Fix regression in `GeoDataFrame.explode()` with non-default geometry column name.
- Fix regression in `apply()` causing row-wise all nan float columns to be casted to GeometryDtype (#2482).
- Fix a crash in datetime column reading where the file contains mixed timezone offsets (#2479). These will be read as UTC localized values.
- Fix a crash in datetime column reading where the file contains datetimes outside the range supported by [ns] precision (#2505).
- Fix regression in passing the Parquet or Feather format `version` in `to_parquet` and `to_feather`. As a result, the `version` parameter for the `to_parquet` and `to_feather` methods has been replaced with `schema_version`. `version` will be passed directly to underlying feather or parquet writer. `version` will only be used to set `schema_version` if `version` is one of 0.1.0 or 0.4.0 (#2496).

# Version 0.11 (June 20, 2022)

Highlights of this release:

- The `geopandas.read_file()` and `GeoDataFrame.to_file()` methods to read and write GIS file formats can now optionally use the `pyogrio` package under the hood through the `engine="pyogrio"` keyword. The pyogrio package implements vectorized IO for GDAL/OGR vector data sources, and is faster compared to the `fiona`-based engine (#2225).
- GeoParquet support updated to implement [v0.4.0](#) of the OpenGeospatial/GeoParquet specification (#2441). Backwards compatibility with v0.1.0 of the metadata spec (implemented in the previous releases of GeoPandas) is guaranteed, and reading and writing Parquet and Feather files will no longer produce a `UserWarning` (#2327).

New features and improvements:

- Improved handling of GeoDataFrame when the active geometry column is lost from the GeoDataFrame. Previously, square bracket indexing `gdf[[...]]` returned a GeoDataFrame when the active geometry column was retained and a DataFrame was returned otherwise. Other pandas indexing methods (`loc`, `iloc`, etc) did not follow the same rules. The new behaviour for all indexing/reshaping operations is now as follows (#2329, #2060):
  - If operations produce a `DataFrame` containing the active geometry column, a GeoDataFrame is returned

- If operations produce a `DataFrame` containing `GeometryDtype` columns, but not the active geometry column, a `GeoDataFrame` is returned, where the active geometry column is set to `None` (set the new geometry column with `set_geometry()`)
- If operations produce a `DataFrame` containing no `GeometryDtype` columns, a `DataFrame` is returned (this can be upcast again by calling `set_geometry()` or the `GeoDataFrame` constructor)
- If operations produce a `Series` of `GeometryDtype`, a `GeoSeries` is returned, otherwise `Series` is returned.
- Error messages for having an invalid geometry column have been improved, indicating the name of the last valid active geometry column set and whether other geometry columns can be promoted to the active geometry column (#2329).
- Datetime fields are now read and written correctly for GIS formats which support them (e.g. GPKG, GeoJSON) with fiona 1.8.14 or higher. Previously, datetimes were read as strings (#2202).
- `folium.Map` keyword arguments can now be specified as the `map_kwds` argument to `GeoDataFrame.explore()` method (#2315).
- Add a new parameter `style_function` to `GeoDataFrame.explore()` to enable plot styling based on GeoJSON properties (#2377).
- It is now possible to write an empty `GeoDataFrame` to a file for supported formats (#2240). Attempting to do so will now emit a `UserWarning` instead of a `ValueError`.
- Fast rectangle clipping has been exposed as `GeoSeries/GeoDataFrame.clip_by_rect()` (#1928).
- The `mask` parameter of `GeoSeries/GeoDataFrame.clip()` now accepts a rectangular mask as a list-like to perform fast rectangle clipping using the new `GeoSeries/GeoDataFrame.clip_by_rect()` (#2414).
- Bundled demo dataset `natural_earth_lowres` has been updated to version 5.0.1 of the source, with field `ISO_A3` manually corrected for some cases (#2418).

Deprecations and compatibility notes:

- The active development branch of geopandas on GitHub has been renamed from master to main (#2277).
- Deprecated methods `GeometryArray.equals_exact()` and `GeometryArray.almost_equals()` have been removed. They should be replaced with `GeometryArray.geom_equals_exact()` and `GeometryArray.geom_almost_equals()` respectively (#2267).
- Deprecated CRS functions `explicit_crs_from_epsg()`, `epsg_from_crs()` and `get_epsg_file_contents()` were removed (#2340).
- Warning about the behaviour change to `GeoSeries.isna()` with empty geometries present has been removed (#2349).
- Specifying a CRS in the `GeoDataFrame/GeoSeries` constructor which contradicted the underlying `GeometryArray` now raises a `ValueError` (#2100).
- Specifying a CRS in the `GeoDataFrame` constructor when no geometry column is provided and calling `GeoDataFrame.set_crs` on a `GeoDataFrame` without an active geometry column now raise a `ValueError` (#2100).

- Passing non-geometry data to the `GeoSeries` constructor is now fully deprecated and will raise a `TypeError` (#2314). Previously, a `pandas.Series` was returned for non-geometry data.
- Deprecated `GeoSeries/GeoDataFrame` set operations `__xor__()`, `__or__()`, `__and__()` and `__sub__()`, `geopandas.io.file.read_file`/`to_file` and `geopandas.io.sql.read_postgis` now emit `FutureWarning` instead of `DeprecationWarning` and will be completely removed in a future release.
- Accessing the `crs` of a `GeoDataFrame` without active geometry column is deprecated and will be removed in GeoPandas 0.12 (#2373).

Bug fixes:

- `GeoSeries.to_frame` now creates a `GeoDataFrame` with the geometry column name set correctly (#2296)
- Fix pickle files created with pygeos installed can not be readable when pygeos is not installed (#2237).
- Fixed `UnboundLocalError` in `GeoDataFrame.plot()` using `legend=True` and `missing_kwds` (#2281).
- Fix `explode()` incorrectly relating index to columns, including where the input index is not unique (#2292)
- Fix `GeoSeries.[xyz]` raising an `IndexError` when the underlying GeoSeries contains empty points (#2335). Rows corresponding to empty points now contain `np.nan`.
- Fix `GeoDataFrame.iloc` raising a `TypeError` when indexing a `GeoDataFrame` with only a single column of `GeometryDtype` (#1970).
- Fix `GeoDataFrame.iterfeatures()` not returning features with the same field order as `GeoDataFrame.columns` (#2396).
- Fix `GeoDataFrame.from_features()` to support reading GeoJSON with null properties (#2243).
- Fix `GeoDataFrame.to_parquet()` not intercepting `engine` keyword argument, breaking consistency with pandas (#2227)
- Fix `GeoDataFrame.explore()` producing an error when `column` is of boolean dtype (#2403).
- Fix an issue where `GeoDataFrame.to_postgis()` output the wrong SRID for ESRI authority CRS (#2414).
- Fix `GeoDataFrame.from_dict/from_features` classmethods using `GeoDataFrame` rather than `cls` as the constructor.
- Fix `GeoDataFrame.plot()` producing incorrect colors with mixed geometry types when `colors` keyword is provided. (#2420)

Notes on (optional) dependencies:

- GeoPandas 0.11 drops support for Python 3.7 and pandas 0.25 (the minimum supported pandas version is now 1.0.5). Further, the minimum required versions for the listed dependencies have now

## Version 0.10.2 (October 16, 2021)

Small bug-fix release:

- Fix regression in `overlay()` in case no geometries are intersecting (but have overlapping total bounds) (#2172).
- Fix regression in `overlay()` with `keep_geom_type=True` in case the overlay of two geometries in a `GeometryCollection` with other geometry types (#2177).
- Fix `overlay()` to honor the `keep_geom_type` keyword for the `op="difference"` case (#2164).
- Fix regression in `plot()` with a `mapclassify` `scheme` in case the formatted legend labels have duplicates (#2166).
- Fix a bug in the `explore()` method ignoring the `vmin` and `vmax` keywords in case they are set to 0 (#2175).
- Fix `unary_union` to correctly handle a `GeoSeries` with missing values (#2181).
- Avoid internal deprecation warning in `clip()` (#2179).

## Version 0.10.1 (October 8, 2021)

Small bug-fix release:

- Fix regression in `overlay()` with non-overlapping geometries and a non-default `how` (i.e. not “intersection”) (#2157).

## Version 0.10.0 (October 3, 2021)

Highlights of this release:

- A new `sjoin_nearest()` method to join based on proximity, with the ability to set a maximum search radius (#1865). In addition, the `sindex` attribute gained a new method for a “nearest” spatial index query (#1865, #2053).
- A new `explore()` method on `GeoDataFrame` and `GeoSeries` with native support for interactive visualization based on folium / leaflet.js (#1953)
- The `geopandas.sjoin() / overlay() / clip()` functions are now also available as methods on the `GeoDataFrame` (#2141, #1984, #2150).

New features and improvements:

- Add support for pandas' `value_counts()` method for geometry dtype (#2047).
- The `explode()` method has a new `ignore_index` keyword (consistent with pandas' explode method) to reset the index in the result, and a new `index_parts` keywords to control whether a cumulative count indexing the parts of the exploded multi-geometries should be added (#1871).
- `points_from_xy()` is now available as a GeoSeries method `from_xy` (#1936).
- The `to_file()` method will now attempt to detect the driver (if not specified) based on the extension of the provided filename, instead of defaulting to ESRI Shapefile (#1609).
- Support for the `storage_options` keyword in `read_parquet()` for specifying filesystem-specific options (e.g. for S3) based on fsspec (#2107).
- The read/write functions now support `~` (user home directory) expansion (#1876).
- Support the `convert_dtypes()` method from pandas to preserve the GeoDataFrame class (#2115).
- Support WKB values in the hex format in `GeoSeries.from_wkb()` (#2106).
- Update the `estimate_utm_crs()` method to handle crossing the antimeridian with pyproj 3.1+ (#2049).
- Improved heuristic to decide how many decimals to show in the repr based on whether the CRS is projected or geographic (#1895).
- Switched the default for `geocode()` from GeoCode.Farm to the Photon geocoding API (<https://photon.komoot.io>) (#2007).

Deprecations and compatibility notes:

- The `op=` keyword of `sjoin()` to indicate which spatial predicate to use for joining is being deprecated and renamed in favor of a new `predicate=` keyword (#1626).
- The `cascaded_union` attribute is deprecated, use `unary_union` instead (#2074).
- Constructing a GeoDataFrame with a duplicated "geometry" column is now disallowed. This can also raise an error in the `pd.concat(..., axis=1)` function if this results in duplicated active geometry columns (#2046).
- The `explode()` method currently returns a GeoSeries/GeoDataFrame with a MultiIndex, with an additional level with indices of the parts of the exploded multi-geometries. For consistency with pandas, this will change in the future and the new `index_parts` keyword is added to control this.

Bug fixes:

- Fix in the `clip()` function to correctly clip MultiPoints instead of leaving them intact when partly outside of the clip bounds (#2148).
- Fix `GeoSeries.isna()` to correctly return a boolean Series in case of an empty GeoSeries (#2073).
- Fix the GeoDataFrame constructor to preserve the geometry name when the argument is already a GeoDataFrame object (i.e. `GeoDataFrame(gdf)`) (#2138).
- Fix loss of the values' CRS when setting those values as a column (`GeoDataFrame.__setitem__`) (#1963)
- Fix in `GeoDataFrame.apply()` to preserve the active geometry column name (#1955).

- Fix in `sjoin()` to not ignore the suffixes in case of a right-join (`how="right"`) (#2065).
- Fix `GeoDataFrame.explode()` with a MultiIndex (#1945).
- Fix the handling of missing values in `to/from_wkb` and `to_from_wkt` (#1891).
- Fix `to_file()` and `to_json()` when DataFrame has duplicate columns to raise an error (#1900).
- Fix bug in the colors shown with user-defined classification scheme (#2019).
- Fix handling of the `path_effects` keyword in `plot()` (#2127).
- Fix `GeoDataFrame.explode()` to preserve `attrs` (#1935)

Notes on (optional) dependencies:

- GeoPandas 0.10.0 dropped support for Python 3.6 and pandas 0.24. Further, the minimum required versions are numpy 1.18, shapely 1.6, fiona 1.8, matplotlib 3.1 and pyproj 2.2.
- Plotting with a classification schema now requires mapclassify version  $\geq 2.4$  (#1737).
- Compatibility fixes for the latest numpy in combination with Shapely 1.7 (#2072)
- Compatibility fixes for the upcoming Shapely 1.8 (#2087).
- Compatibility fixes for the latest PyGEOS (#1872, #2014) and matplotlib (colorbar issue, #2066).

## Version 0.9.0 (February 28, 2021)

Many documentation improvements and a restyled and restructured website with a new logo (#1564, #1579, #1617, #1668, #1731, #1750, #1757, #1759).

New features and improvements:

- The `geopandas.read_file` function now accepts more general file-like objects (e.g. `fsspec` open file objects). It will now also automatically recognize zipped files (#1535).
- The `GeoDataFrame.plot()` method now provides access to the pandas plotting functionality for the non-geometry columns, either using the `kind` keyword or the accessor method (e.g. `gdf.plot(kind="bar")` or `gdf.plot.bar()`) (#1465).
- New `from_wkt()`, `from_wkb()`, `to_wkt()`, `to_wkb()` methods for GeoSeries to construct a GeoSeries from geometries in WKT or WKB representation, or to convert a GeoSeries to a pandas Series with WKT or WKB values (#1710).
- New `GeoSeries.z` attribute to access the z-coordinates of Point geometries (similar to the existing `.x` and `.y` attributes) (#1773).
- The `to_crs()` method now handles missing values (#1618).
- Support for pandas' new `.attrs` functionality (#1658).
- The `dissolve()` method now allows dissolving by no column (`by=None`) to create a union of all geometries (single-row GeoDataFrame) (#1568).
- New `estimate_utm_crs()` method on GeoSeries/GeoDataFrame to determine the UTM CRS based on the bounds (#1646).

- `GeoDataFrame.from_dict()` now accepts `geometry` and `crs` keywords (#1619).
- `GeoDataFrame.to_postgis()` and `geopandas.read_postgis()` now supports both sqlalchemy engine and connection objects (#1638).
- The `GeoDataFrame.explode()` method now allows exploding based on a non-geometry column, using the pandas implementation (#1720).
- Performance improvement in `GeoDataFrame/GeoSeries.explode()` when using the PyGEOS backend (#1693).
- The binary operation and predicate methods (eg `intersection()`, `intersects()`) have a new `align` keyword which allows optionally not aligning on the index before performing the operation with `align=False` (#1668).
- The `GeoDataFrame.dissolve()` method now supports all relevant keywords of `groupby()`, i.e. the `level`, `sort`, `observed` and `dropna` keywords (#1845).
- The `geopandas.overlay()` function now accepts `make_valid=False` to skip the step to ensure the input geometries are valid using `buffer(0)` (#1802).
- The `GeoDataFrame.to_json()` method gained a `drop_id` keyword to optionally not write the GeoDataFrame's index as the "id" field in the resulting JSON (#1637).
- A new `aspect` keyword in the plotting methods to optionally allow retaining the original aspect (#1512).
- A new `interval` keyword in the `legend_kwds` group of the `plot()` method to control the appearance of the legend labels when using a classification scheme (#1605).
- The spatial index of a GeoSeries (accessed with the `sindex` attribute) is now stored on the underlying array. This ensures that the spatial index is preserved in more operations where possible, and that multiple geometry columns of a GeoDataFrame can each have a spatial index (#1444).
- Addition of a `has_sindex` attribute on the GeoSeries/GeoDataFrame to check if a spatial index has already been initialized (#1627).
- The `geopandas.testing.assert_geoseries_equal()` and `assert_geodataframe_equal()` testing utilities now have a `normalize` keyword (False by default) to normalize geometries before comparing for equality (#1826). Those functions now also give a more informative error message when failing (#1808).

Deprecations and compatibility notes:

- The `is_ring` attribute currently returns True for Polygons. In the future, this will be False (#1631). In addition, start to check it for LineStrings and LinearRings (instead of always returning False).
- The deprecated `objects` keyword in the `intersection()` method of the `GeoDataFrame/GeoSeries.sindex` spatial index object has been removed (#1444).

Bug fixes:

- Fix regression in the `plot()` method raising an error with empty geometries (#1702, #1828).
- Fix `geopandas.overlay()` to preserve geometries of the correct type which are nested within a GeometryCollection as a result of the overlay operation (#1582). In addition, a warning will now be

raised if geometries of different type are dropped from the result (#1554).

- Fix the repr of an empty GeoSeries to not show spurious warnings (#1673).
- Fix the `.crs` for empty GeoDataFrames (#1560).
- Fix `geopandas.clip` to preserve the correct geometry column name (#1566).
- Fix bug in `plot()` method when using `legend_kwds` with multiple subplots (#1583)
- Fix spurious warning with `missing_kwds` keyword of the `plot()` method when there are no areas with missing data (#1600).
- Fix the `plot()` method to correctly align values passed to the `column` keyword as a pandas Series (#1670).
- Fix bug in plotting MultiPoints when passing values to determine the color (#1694)
- The `rename_geometry()` method now raises a more informative error message when a duplicate column name is used (#1602).
- Fix `explode()` method to preserve the CRS (#1655)
- Fix the `GeoSeries.apply()` method to again accept the `convert_dtype` keyword to be consistent with pandas (#1636).
- Fix `GeoDataFrame.apply()` to preserve the CRS when possible (#1848).
- Fix bug in containment test as `geom in geoseries` (#1753).
- The `shift()` method of a GeoSeries/GeoDataFrame now preserves the CRS (#1744).
- The PostGIS IO functionality now quotes table names to ensure it works with case-sensitive names (#1825).
- Fix the `GeoSeries` constructor without passing data but only an index (#1798).

Notes on (optional) dependencies:

- GeoPandas 0.9.0 dropped support for Python 3.5. Further, the minimum required versions are pandas 0.24, numpy 1.15 and shapely 1.6 and fiona 1.8.
- The `descartes` package is no longer required for plotting polygons. This functionality is now included by default in GeoPandas itself, when matplotlib is available (#1677).
- Fiona is now only imported when used in `read_file/to_file`. This means you can now force geopandas to install without fiona installed (although it is still a default requirement) (#1775).
- Compatibility with the upcoming Shapely 1.8 (#1659, #1662, #1819).

## Version 0.8.2 (January 25, 2021)

Small bug-fix release for compatibility with PyGEOS 0.9.

## Version 0.8.1 (July 15, 2020)

Small bug-fix release:

- Fix a regression in the `plot()` method when visualizing with a JenksCaspallSampled or FisherJenksSampled scheme (#1486).
- Fix spurious warning in `GeoDataFrame.to_postgis` (#1497).
- Fix the un-pickling with `pd.read_pickle` of files written with older GeoPandas versions (#1511).

## Version 0.8.0 (June 24, 2020)

**Experimental:** optional use of PyGEOS to speed up spatial operations (#1155). PyGEOS is a faster alternative for Shapely (being contributed back to a future version of Shapely), and is used in element-wise spatial operations and for spatial index in e.g. `sjoin` (#1343, #1401, #1421, #1427, #1428). See the [installation docs](#) for more info and how to enable it.

New features and improvements:

- IO enhancements:
  - New `GeoDataFrame.to_postgis()` method to write to PostGIS database (#1248).
  - New Apache Parquet and Feather file format support (#1180, #1435)
  - Allow appending to files with `GeoDataFrame.to_file` (#1229).
  - Add support for the `ignore_geometry` keyword in `read_file` to only read the attribute data. If set to True, a pandas DataFrame without geometry is returned (#1383).
  - `geopandas.read_file` now supports reading from file-like objects (#1329).
  - `GeoDataFrame.to_file` now supports specifying the CRS to write to the file (#802). By default it still uses the CRS of the GeoDataFrame.
  - New `chunksize` keyword in `geopandas.read_postgis` to read a query in chunks (#1123).
- Improvements related to geometry columns and CRS:
  - Any column of the GeoDataFrame that has a “geometry” dtype is now returned as a GeoSeries. This means that when having multiple geometry columns, not only the “active” geometry column is returned as a GeoSeries, but also accessing another geometry column (`gdf["other_geom_column"]`) gives a GeoSeries (#1336).
  - Multiple geometry columns in a GeoDataFrame can now each have a different CRS. The global `gdf.crs` attribute continues to returns the CRS of the “active” geometry column. The CRS of other geometry columns can be accessed from the column itself (eg `gdf["other_geom_column"].crs`) (#1339).
  - New `set_crs()` method on GeoDataFrame/GeoSeries to set the CRS of naive geometries (#747).
- Improvements related to plotting:
  - The y-axis is now scaled depending on the center of the plot when using a geographic CRS, instead of using an equal aspect ratio (#1290).
  - When passing a column of categorical dtype to the `column=` keyword of the GeoDataFrame `plot()`, we now honor all categories and its order (#1483). In addition, a new `categories`

keyword allows to specify all categories and their order otherwise (#1173).

- For choropleths using a classification scheme (using `scheme=`), the `legend_kwds` accept two new keywords to control the formatting of the legend: `fmt` with a format string for the bin edges (#1253), and `labels` to pass fully custom class labels (#1302).
- New `covers()` and `covered_by()` methods on GeoSeries/GeoDataframe for the equivalent spatial predicates (#1460, #1462).
- GeoPandas now warns when using distance-based methods with data in a geographic projection (#1378).

## Deprecations:

- When constructing a GeoSeries or GeoDataFrame from data that already has a CRS, a deprecation warning is raised when both CRS don't match, and in the future an error will be raised in such a case. You can use the new `set_crs` method to override an existing CRS. See [the docs](#).
- The helper functions in the `geopandas.plotting` module are deprecated for public usage (#656).
- The `geopandas.io` functions are deprecated, use the top-level `read_file` and `to_file` instead (#1407).
- The set operators (`&`, `|`, `^`, `-`) are deprecated, use the `intersection()`, `union()`, `symmetric_difference()`, `difference()` methods instead (#1255).
- The `sindex` for empty dataframe will in the future return an empty spatial index instead of `None` (#1438).
- The `objects` keyword in the `intersection` method of the spatial index returned by the `sindex` attribute is deprecated and will be removed in the future (#1440).

## Bug fixes:

- Fix the `total_bounds()` method to ignore missing and empty geometries (#1312).
- Fix `geopandas.clip` when masking with non-overlapping area resulting in an empty GeoDataFrame (#1309, #1365).
- Fix error in `geopandas.sjoin` when joining on an empty geometry column (#1318).
- CRS related fixes: `pandas.concat` preserves CRS when concatenating GeoSeries objects (#1340), preserve the CRS in `geopandas.clip` (#1362) and in `GeoDataFrame.astype` (#1366).
- Fix bug in `GeoDataFrame.explode()` when 'level\_1' is one of the column names (#1445).
- Better error message when rtree is not installed (#1425).
- Fix bug in `GeoSeries.equals()` (#1451).
- Fix plotting of multi-part geometries with additional style keywords (#1385).

And we now have a [Code of Conduct!](#)

GeoPandas 0.8.0 is the last release to support Python 3.5. The next release will require Python 3.6, pandas 0.24, numpy 1.15 and shapely 1.6 or higher.

# Version 0.7.0 (February 16, 2020)

Support for Python 2.7 has been dropped. GeoPandas now works with Python >= 3.5.

The important API change of this release is that GeoPandas now requires PROJ > 6 and pyproj > 2.2, and that the `.crs` attribute of a GeoSeries and GeoDataFrame no longer stores the CRS information as a proj4 string or dict, but as a `pyproj.CRS` object (#1101).

This gives a better user interface and integrates improvements from pyproj and PROJ 6, but might also require some changes in your code. Check the [migration guide](#) in the documentation.

Other API changes;

- The `GeoDataFrame.to_file` method will now also write the GeoDataFrame index to the file, if the index is named and/or non-integer. You can use the `index=True/False` keyword to overwrite this default inference (#1059).

New features and improvements:

- A new `geopandas.clip` function to clip a GeoDataFrame to the spatial extent of another shape (#1128).
- The `geopandas.overlay` function now works for all geometry types, including points and linestrings in addition to polygons (#1110).
- The `plot()` method gained support for missing values (in the column that determines the colors). By default it doesn't plot the corresponding geometries, but using the new `missing_kwds` argument you can specify how to style those geometries (#1156).
- The `plot()` method now also supports plotting GeometryCollection and LinearRing objects (#1225).
- Added support for filtering with a geometry or reading a subset of the rows in `geopandas.read_file` (#1160).
- Added support for the new nullable integer data type of pandas in `GeoDataFrame.to_file` (#1220).

Bug fixes:

- `GeoSeries.reset_index()` now correctly results in a GeoDataFrame instead of DataFrame (#1252).
- Fixed the `geopandas.sjoin` function to handle MultiIndex correctly (#1159).
- Fixed the `geopandas.sjoin` function to preserve the index name of the left GeoDataFrame (#1150).

# Version 0.6.3 (February 6, 2020)

Small bug-fix release:

- Compatibility with Shapely 1.7 and pandas 1.0 (#1244).

- Fix `GeoDataFrame.fillna` to accept non-geometry values again when there are no missing values in the geometry column. This should make it easier to fill the numerical columns of the GeoDataFrame (#1279).

## Version 0.6.2 (November 18, 2019)

Small bug-fix release fixing a few regressions:

- Fix a regression in passing an array of RRB(A) tuples to the `.plot()` method (#1178, #1211).
- Fix the `bounds` and `total_bounds` attributes for empty GeoSeries, which also fixes the repr of an empty or all-NA GeoSeries (#1184, #1195).
- Fix filtering of a GeoDataFrame to preserve the index type when ending up with an empty result (#1190).

## Version 0.6.1 (October 12, 2019)

Small bug-fix release fixing a few regressions:

- Fix `astype` when converting to string with Multi geometries (#1145) or when converting a dataframe without geometries (#1144).
- Fix `GeoSeries.fillna` to accept `np.nan` again (#1149).

## Version 0.6.0 (September 27, 2019)

Important note! This will be the last release to support Python 2.7 (#1031)

API changes:

- A refactor of the internals based on the pandas ExtensionArray interface (#1000). The main user visible changes are:
  - The `.dtype` of a GeoSeries is now a `'geometry'` dtype (and no longer a numpy `object` dtype).
  - The `.values` of a GeoSeries now returns a custom `GeometryArray`, and no longer a numpy array. To get back a numpy array of Shapely scalars, you can convert explicitly using `np.asarray(...)`.
- The `GeoSeries` constructor now raises a warning when passed non-geometry data. Currently the constructor falls back to return a pandas `Series`, but in the future this will raise an error (#1085).
- The missing value handling has been changed to now separate the concepts of missing geometries and empty geometries (#601, 1062). In practice this means that (see [the docs](#) for more details):

- `GeoSeries.isna` now considers only missing values, and if you want to check for empty geometries, you can use `GeoSeries.is_empty` (`GeoDataFrame.isna` already only looked at missing values).
- `GeoSeries.dropna` now actually drops missing values (before it didn't drop either missing or empty geometries)
- `GeoSeries.fillna` only fills missing values (behaviour unchanged).
- `GeoSeries.align` uses missing values instead of empty geometries by default to fill non-matching index entries.

New features and improvements:

- Addition of a `GeoSeries.affine_transform` method, equivalent of Shapely's function (#1008).
- Addition of a `GeoDataFrame.rename_geometry` method to easily rename the active geometry column (#1053).
- Addition of `geopandas.show_versions()` function, which can be used to give an overview of the installed libraries in bug reports (#899).
- The `legend_kwds` keyword of the `plot()` method can now also be used to specify keywords for the color bar (#1102).
- Performance improvement in the `sjoin()` operation by re-using existing spatial index of the input dataframes, if available (#789).
- Updated documentation to work with latest version of geoplot and contextily (#1044, #1088).
- A new `geopandas.options` configuration, with currently a single option to control the display precision of the coordinates (`options.display_precision`). The default is now to show less coordinates (3 for projected and 5 for geographic coordinates), but the default can be overridden with the option.

Bug fixes:

- Also try to use `pysal` instead of `mapclassify` if available (#1082).
- The `GeoDataFrame.astype()` method now correctly returns a `GeoDataFrame` if the geometry column is preserved (#1009).
- The `to_crs` method now uses `always_xy=True` to ensure correct lon/lat order handling for pyproj>=2.2.0 (#1122).
- Fixed passing list-like colors in the `plot()` method in case of "multi" geometries (#1119).
- Fixed the coloring of shapes and colorbar when passing a custom `norm` in the `plot()` method (#1091, #1089).
- Fixed `GeoDataFrame.to_file` to preserve VFS file paths (e.g. when a "s3://" path is specified) (#1124).
- Fixed failing case in `geopandas.sjoin` with empty geometries (#1138).

In addition, the minimum required versions of some dependencies have been increased: GeoPandas now requires pandas >=0.23.4 and matplotlib >=2.0.1 (#1002).

# Version 0.5.1 (July 11, 2019)

- Compatibility with latest mapclassify version 2.1.0 (#1025).

# Version 0.5.0 (April 25, 2019)

## Improvements:

- Significant performance improvement (around 10x) for `GeoDataFrame.iterfeatures`, which also improves `GeoDataFrame.to_file` (#864).
- File IO enhancements based on Fiona 1.8:
  - Support for writing bool dtype (#855) and datetime dtype, if the file format supports it (#728).
  - Support for writing dataframes with multiple geometry types, if the file format allows it (e.g. GeoJSON for all types, or ESRI Shapefile for Polygon+MultiPolygon) (#827, #867, #870).
- Compatibility with pyproj >= 2 (#962).
- A new `geopandas.points_from_xy()` helper function to convert x and y coordinates to Point objects (#896).
- The `buffer` and `interpolate` methods now accept an array-like to specify a variable distance for each geometry (#781).
- Addition of a `relate` method, corresponding to the shapely method that returns the DE-9IM matrix (#853).
- Plotting improvements:
  - Performance improvement in plotting by only flattening the geometries if there are actually ‘Multi’ geometries (#785).
  - Choropleths: access to all `mapclassify` classification schemes and addition of the `classification_kwds` keyword in the `plot` method to specify options for the scheme (#876).
  - Ability to specify a matplotlib axes object on which to plot the color bar with the `cax` keyword, in order to have more control over the color bar placement (#894).
- Changed the default provider in `geopandas.tools.geocode` from Google (now requires an API key) to Geocode.Farm (#907, #975).

## Bug fixes:

- Remove the edge in the legend marker (#807).
- Fix the `align` method to preserve the CRS (#829).
- Fix `geopandas.testing.assert_geodataframe_equal` to correctly compare left and right dataframes (#810).
- Fix in choropleth mapping when the values contain missing values (#877).

- Better error message in `sjoin` if the input is not a GeoDataFrame (#842).
- Fix in `read_postgis` to handle nullable (missing) geometries (#856).
- Correctly passing through the `parse_dates` keyword in `read_postgis` to the underlying pandas method (#860).
- Fixed the shape of Antarctica in the included demo dataset ‘naturalearth\_lowres’ (by updating to the latest version) (#804).

## Version 0.4.1 (March 5, 2019)

Small bug-fix release for compatibility with the latest Fiona and PySAL releases:

- Compatibility with Fiona 1.8: fix deprecation warning (#854).
- Compatibility with PySAL 2.0: switched to `mapclassify` instead of `PySAL` as dependency for choropleth mapping with the `scheme` keyword (#872).
- Fix for new `overlay` implementation in case the intersection is empty (#800).

## Version 0.4.0 (July 15, 2018)

Improvements:

- Improved `overlay` function (better performance, several incorrect behaviours fixed) (#429)
- Pass keywords to control legend behavior (`legend_kwds`) to `plot` (#434)
- Add basic support for reading remote datasets in `read_file` (#531)
- Pass kwargs for `buffer` operation on GeoSeries (#535)
- Expose all geopy services as options in geocoding (#550)
- Faster write speeds to GeoPackage (#605)
- Permit `read_file` filtering with a bounding box from a GeoDataFrame (#613)
- Set CRS on GeoDataFrame returned by `read_postgis` (#627)
- Permit setting markersize for Point GeoSeries plots with column values (#633)
- Started an example gallery (#463, #690, #717)
- Support for plotting MultiPoints (#683)
- Testing functionality (e.g. `assert_geodataframe_equal`) is now publicly exposed (#707)
- Add `explode` method to GeoDataFrame (similar to the GeoSeries method) (#671)
- Set equal aspect on active axis on multi-axis figures (#718)
- Pass array of values to column argument in `plot` (#770)

Bug fixes:

- Ensure that colorbars are plotted on the correct axis (#523)

- Handle plotting empty GeoDataFrame (#571)
- Save z-dimension when writing files (#652)
- Handle reading empty shapefiles (#653)
- Correct dtype for empty result of spatial operations (#685)
- Fix empty `sjoin` handling for pandas $\geq 0.23$  (#762)

## Version 0.3.0 (August 29, 2017)

Improvements:

- Improve plotting performance using `matplotlib.collections` (#267)
- Improve default plotting appearance. The defaults now follow the new matplotlib defaults (#318, #502, #510)
- Provide access to x/y coordinates as attributes for Point GeoSeries (#383)
- Make the NYBB dataset available through `geopandas.datasets` (#384)
- Enable `sjoin` on non-integer-index GeoDataFrames (#422)
- Add `cx` indexer to GeoDataFrame (#482)
- `GeoDataFrame.from_features` now also accepts a Feature Collection (#225, #507)
- Use index label instead of integer id in output of `iterfeatures` and `to_json` (#421)
- Return empty data frame rather than raising an error when performing a spatial join with non overlapping geodataframes (#335)

Bug fixes:

- Compatibility with shapely 1.6.0 (#512)
- Fix `fiona.filter` results when bbox is not None (#372)
- Fix `dissolve` to retain CRS (#389)
- Fix `cx` behavior when using index of 0 (#478)
- Fix display of lower bin in legend label of choropleth plots using a PySAL scheme (#450)

## Version 0.2.0

Improvements:

- Complete overhaul of the documentation
- Addition of `overlay` to perform spatial overlays with polygons (#142)
- Addition of `sjoin` to perform spatial joins (#115, #145, #188)
- Addition of `__geo_interface__` that returns a python data structure to represent the `GeoSeries` as a GeoJSON-like `FeatureCollection` (#116) and `iterfeatures` method (#178)

- Addition of the `explode` (#146) and `dissolve` (#310, #311) methods.
- Addition of the `sindex` attribute, a Spatial Index using the optional dependency `rtree` (`libspatialindex`) that can be used to speed up certain operations such as overlays (#140, #141).
- Addition of the `GeoSeries.cx` coordinate indexer to slice a GeoSeries based on a bounding box of the

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme 0.15.4](#).

Created using [Sphinx 7.3.7](#).

The documentation of GeoPandas consists of four parts - :doc:`User Guide <docs/user\_guide>` with explanation of the basic functionality, :doc:`Advanced Guide <docs/advanced\_guide>` covering topics which assume knowledge of basics, :doc:`Examples <gallery/index>`, and :doc:`API reference <docs/reference>` detailing every class, method, function and attribute used implemented by GeoPandas.

.. container:: button

:doc:`User guide <docs/user\_guide>` :doc:`Advanced guide <docs/advanced\_guide>`  
:doc:`Examples <gallery/index>` :doc:`API reference <docs/reference>`

.. toctree::

:maxdepth: 2

:caption: Documentation

User guide <docs/user\_guide>  
Advanced guide <docs/advanced\_guide>  
API reference <docs/reference>  
Changelog <docs/changelog>

# 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 [GitHub Actions](#). Status on GHA will be visible on a pull request. GHA are automatically enabled on your own fork as well. To trigger a check, make a PR to your own fork.
- 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 documented explicitly.
- Follow PEP 8 when possible. We use [Black](#) and [ruff](#) to ensure a consistent code format throughout the project. For more details see [below](#).
- 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.
- GeoPandas supports Python 3.9+ only. The last version of GeoPandas supporting Python 2 is 0.6.
- Unless your PR implements minor changes or internal work only, make sure it contains a note describing the changes in the *CHANGELOG.md* file.

## 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 GitHub Actions once your pull request is submitted. The test suite will also automatically run on your branch so you can check it prior to submitting the pull request.

# Creating a branch

You want your main 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 main branch:

```
git fetch upstream
git rebase upstream/main
```

This will replay your commits on top of the latest GeoPandas git main. 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

## Using the provided environment

GeoPandas provides an environment which includes the required dependencies for development. The environment file is located in the top level of the repo and is named `environment-dev.yml`. You can create this environment by navigating to the the `geopandas` source directory and running:

```
conda env create -f environment-dev.yml
```

This will create a new conda environment named `geopandas_dev`.

## Creating the environment manually

Alternatively, it is possible to create a development environment manually. To do this, 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 python
```

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

## Working with the environment

To work in this environment, you need to `activate` it. The instructions below should work for both Windows, Mac and Linux:

```
conda activate geopandas_dev
```

Once your environment is activated, you will 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:

```
conda 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 a development environment, you must first install the dependencies of GeoPandas. If you used the provided environment in section 2, skip this step and continue to section 4. If you created the environment manually, we suggest installing dependencies using the following commands (executed after your development environment has been activated):

```
conda install -c conda-forge pandas pyogrio shapely pyproj pytest
```

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 -m pip install -e .
```

## 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.

GeoPandas uses the [pytest 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:

```
pytest
```

## 6) Updating the Documentation

GeoPandas documentation resides in the `doc` folder. Changes to the docs are made by modifying the appropriate file in the `source` folder within `doc`. GeoPandas docs use mixture of reStructuredText syntax for `rst` files, [which is explained here](#) and MyST syntax for `md` files [explained here](#). The docstrings follow the [Numpy Docstring standard](#). Some pages and examples are Jupyter notebooks converted to docs using [nbsphinx](#). Jupyter notebooks should be stored without the output.

We highly encourage you to follow the [Google developer documentation style guide](#) when updating or creating new documentation.

Once you have made your changes, you may try if they render correctly by building the docs using sphinx. To do so, you can navigate to the `doc` folder:

```
cd doc
```

and type:

```
make html
```

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

In case of any errors, you can try to use `make html` within a new environment based on `environment.yml` specification in the `doc` folder. You may need to register Jupyter kernel as `geopandas_docs`. Using conda:

```
cd doc
conda env create -f environment.yml
conda activate geopandas_docs
python -m ipykernel install --user --name geopandas_docs
make html
```

For minor updates, you can skip the `make html` part as reStructuredText and MyST syntax are usually quite straightforward.

## 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](#).

## Style Guide & Linting

GeoPandas follows the [PEP8](#) standard and uses [Black](#) and [ruff](#) to ensure a consistent code format throughout the project.

Continuous Integration (GitHub Actions) will run those tools and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
black geopandas
git diff upstream/main -u -- "*.py" | ruff .
```

to auto-format your code. Additionally, many editors have plugins that will apply `black` as you edit files.

Optionally (but recommended), you can setup [pre-commit hooks](#) to automatically run `black` and `ruff` when you make a git commit. If you did not use the provided development environment in `environment-dev.yml`, you must first install `pre-commit`:

```
$ python -m pip install pre-commit
```

From the root of the geopandas repository, you should then install the `pre-commit` included in GeoPandas:

```
$ pre-commit install
```

Then `black` and `ruff` will be run automatically each time you commit changes. You can skip these checks with `git commit --no-verify`.

## Commit message conventions

Commit your changes to your local repository with an explanatory message. GeoPandas uses the pandas convention for commit message prefixes and layout. Here are some common prefixes along with general guidelines for when to use them:

- ENH: Enhancement, new functionality
- BUG: Bug fix
- DOC: Additions/updates to documentation
- TST: Additions/updates to tests
- BLD: Updates to the build process/scripts
- PERF: Performance improvement
- TYP: Type annotations
- CLN: Code cleanup

The following defines how a commit message should be structured. Please refer to the relevant GitHub issues in your commit message using GH1234 or #1234. Either style is fine, but the former is generally preferred:

- a subject line with < 80 chars.
- One blank line.
- ~~Optionally a commit message body~~

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.

# GeoPandas Project Code of Conduct

Behind the GeoPandas Project is an engaged and respectful community made up of people from all over the world and with a wide range of backgrounds. Naturally, this implies diversity of ideas and perspectives on often complex problems. Disagreement and healthy discussion of conflicting viewpoints is welcome: the best solutions to hard problems rarely come from a single angle. But disagreement is not an excuse for aggression: humans tend to take disagreement personally and easily drift into behavior that ultimately degrades a community. This is particularly acute with online communication across language and cultural gaps, where many cues of human behavior are unavailable. We are outlining here a set of principles and processes to support a healthy community in the face of these challenges.

Fundamentally, we are committed to fostering a productive, harassment-free environment for everyone. Rather than considering this code an exhaustive list of things that you can't do, take it in the spirit it is intended - a guide to make it easier to enrich all of us and the communities in which we participate.

Importantly: as a member of our community, *you are also a steward of these values*. Not all problems need to be resolved via formal processes, and often a quick, friendly but clear word on an online forum or in person can help resolve a misunderstanding and de-escalate things.

However, sometimes these informal processes may be inadequate: they fail to work, there is urgency or risk to someone, nobody is intervening publicly and you don't feel comfortable speaking in public, etc. For these or other reasons, structured follow-up may be necessary and here we provide the means for that: we welcome reports by emailing [geopandas-conduct@googlegroups.com](mailto:geopandas-conduct@googlegroups.com) or by filling out [this form](#).

This code applies equally to founders, developers, mentors and new community members, in all spaces managed by the GeoPandas Project. This includes the mailing lists, our GitHub organization, our chat room, in-person events, and any other forums created by the project team. In addition, violations of this code outside these spaces may affect a person's ability to participate within them.

By embracing the following principles, guidelines and actions to follow or avoid, you will help us make Jupyter a welcoming and productive community. Feel free to contact the Code of Conduct Committee at [geopandas-conduct@googlegroups.com](mailto:geopandas-conduct@googlegroups.com) with any questions.

- 1. Be friendly and patient.**
- 2. Be welcoming.** We strive to be a community that welcomes and supports people of all backgrounds and identities. This includes, but is not limited to, members of any race, ethnicity, culture, national origin, color, immigration status, social and economic class, educational level, sex, sexual orientation, gender identity and expression, age, physical appearance, family status, technological or professional choices, academic discipline, religion, mental ability, and physical ability.
- 3. Be considerate.** Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions. Remember that we're a world-wide community. You may be communicating with someone with a different primary language or cultural background.

4. **Be respectful.** Not all of us will agree all the time, but disagreement is no excuse for poor behavior or poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one.
5. **Be careful in the words that you choose.** Be kind to others. Do not insult or put down other community members. Harassment and other exclusionary behavior are not acceptable. This includes, but is not limited to:
  - Violent threats or violent language directed against another person
  - Discriminatory jokes and language
  - Posting sexually explicit or violent material
  - Posting (or threatening to post) other people's personally identifying information ("doxing")
  - Personal insults, especially those using racist, sexist, and xenophobic terms
  - Unwelcome sexual attention
  - Advocating for, or encouraging, any of the above behavior
  - Repeated harassment of others. In general, if someone asks you to stop, then stop
6. **Moderate your expectations.** Please respect that community members choose how they spend their time in the project. A thoughtful question about your expectations is preferable to demands for another person's time.
7. **When we disagree, try to understand why.** Disagreements, both social and technical, happen all the time and the GeoPandas Project is no exception. Try to understand where others are coming from, as seeing a question from their viewpoint may help find a new path forward. And don't forget that it is human to err: blaming each other doesn't get us anywhere, while we can learn from mistakes to find better solutions.
8. **A simple apology can go a long way.** It can often de-escalate a situation, and telling someone that you are sorry is an act of empathy that doesn't automatically imply an admission of guilt.

## Reporting

If you believe someone is violating the code of conduct, please report this in a timely manner. Code of conduct violations reduce the value of the community for everyone and we take them seriously.

You can file a report by emailing [geopandas-conduct@googlegroups.com](mailto:geopandas-conduct@googlegroups.com) or by filing out [this form](#).

The online form gives you the option to keep your report anonymous or request that we follow up with you directly. While we cannot follow up on an anonymous report, we will take appropriate action.

Messages sent to the e-mail address or through the form will be sent only to the Code of Conduct Committee, which currently consists of:

- Hannah Aizenman
- Joris Van den Bossche

# Enforcement

Enforcement procedures within the GeoPandas Project follow Project Jupyter's [Enforcement Manual](#). For information on enforcement, please view the [original manual](#).

Original text courtesy of the [Speak Up!](#), [Django](#) and [Jupyter](#) Projects, modified by the GeoPandas Project. We are grateful to those projects for contributing these materials under open licensing terms for us to easily reuse.

All content on this page is licensed under a [Creative Commons Attribution](#) license.

---

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

# Ecosystem

## GeoPandas dependencies

GeoPandas brings together the full capability of `pandas` and the open-source geospatial tools `Shapely`, which brings manipulation and analysis of geometric objects backed by `GEOS` library, `pyogrio`, allowing us to read and write geographic data files using `GDAL`, and `pyproj`, a library for cartographic projections and coordinate transformations, which is a Python interface to `PROJ`.

Furthermore, GeoPandas has several optional dependencies as `mapclassify`, or `geopy`.

## Required dependencies

### pandas

`pandas` is a Python package that provides fast, flexible, and expressive data structures designed to make working with structured (tabular, multidimensional, potentially heterogeneous) and time series data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language. It is already well on its way toward this goal.

### Shapely

`Shapely` is a BSD-licensed Python package for manipulation and analysis of planar geometric objects. It is based on the widely deployed `GEOS` (the engine of PostGIS) and `JTS` (from which `GEOS` is ported) libraries. `Shapely` is not concerned with data formats or coordinate systems, but can be readily integrated with packages that are.

### pyogrio

Pyogrio provides a GeoPandas-oriented API to OGR vector data sources, such as ESRI Shapefile, GeoPackage, and GeoJSON. Vector data sources have geometries, such as points, lines, or polygons, and associated records with potentially many columns worth of data.

### pyproj

`pyproj` is a Python interface to `PROJ` (cartographic projections and coordinate transformations library). GeoPandas uses a `pyproj.crs.CRS` object to keep track of the projection of each `GeoSeries` and its `Transformer` object to manage re-projections.

## Optional dependencies

### mapclassify

`mapclassify` provides functionality for Choropleth map classification. Currently, fifteen different classification schemes are available, including a highly-optimized implementation of Fisher-Jenks optimal classification. Each scheme inherits a common structure that ensures computations are scalable and supports applications in streaming contexts.

### geopy

`geopy` is a Python client for several popular geocoding web services. `geopy` makes it easy for Python developers to locate the coordinates of addresses, cities, countries, and landmarks across the globe using third-party geocoders and other data sources.

### matplotlib

`Matplotlib` is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, web application servers, and various graphical user interface toolkits.

### Fiona

`Fiona` is `GDAL`'s neat and nimble vector API for Python programmers. Fiona is designed to be simple and dependable. It focuses on reading and writing data in standard Python IO style and relies upon familiar Python types and protocols such as files, dictionaries, mappings, and iterators instead of classes specific to `OGR`. Fiona can read and write real-world data using multi-layered GIS formats and zipped virtual file systems and integrates readily with other Python GIS packages such as `pyproj`, `Rtree`, and `Shapely`.

## GeoPandas ecosystem

Various packages are built on top of GeoPandas addressing specific geospatial data processing needs, analysis, and visualization. Below is an incomplete list (in no particular order) of tools which form the GeoPandas-related Python ecosystem.

# Spatial analysis and Machine Learning

## PySAL

**PySAL**, the Python spatial analysis library, is an open source cross-platform library for geospatial data science with an emphasis on geospatial vector data written in Python. **PySAL** is a family of packages, some of which are listed below.

### libpysal

**libpysal** provides foundational algorithms and data structures that support the rest of the library. This currently includes the following modules: input/output (**io**), which provides readers and writers for common geospatial file formats; weights (**weights**), which provides the main class to store spatial weights matrices, as well as several utilities to manipulate and operate on them; computational geometry (**cg**), with several algorithms, such as Voronoi tessellations or alpha shapes that efficiently process geometric shapes; and an additional module with example data sets (**examples**).

### esda

**esda** implements methods for the analysis of both global (map-wide) and local (focal) spatial autocorrelation, for both continuous and binary data. In addition, the package increasingly offers cutting-edge statistics about boundary strength and measures of aggregation error in statistical analyses.

### segregation

**segregation** package calculates over 40 different segregation indices and provides a suite of additional features for measurement, visualization, and hypothesis testing that together represent the state of the art in quantitative segregation analysis.

### mgwr

**mgwr** provides scalable algorithms for estimation, inference, and prediction using single- and multi-scale geographically weighted regression models in a variety of generalized linear model frameworks, as well as model diagnostics tools.

### tobler

**tobler** provides functionality for areal interpolation and dasymetric mapping. **tobler** includes functionality for interpolating data using area-weighted approaches, regression model-based approaches

that leverage remotely-sensed raster data as auxiliary information, and hybrid approaches.

## [movingpandas](#)

[MovingPandas](#) is a package for dealing with movement data. [MovingPandas](#) implements a [Trajectory](#) class and corresponding methods based on GeoPandas. A trajectory has a time-ordered series of point geometries. These points and associated attributes are stored in a [GeoDataFrame](#). [MovingPandas](#) implements spatial and temporal data access and analysis functions as well as plotting functions.

## [momepy](#)

[momepy](#) is a library for quantitative analysis of urban form - urban morphometrics. It is built on top of [GeoPandas](#), [PySAL](#) and [networkX](#). [momepy](#) aims to provide a wide range of tools for a systematic and exhaustive analysis of urban form. It can work with a wide range of elements, while focused on building footprints and street networks.

## [geosnap](#)

[geosnap](#) makes it easier to explore, model, analyze, and visualize the social and spatial dynamics of neighborhoods. [geosnap](#) provides a suite of tools for creating socio-spatial datasets, harmonizing those datasets into consistent set of time-static boundaries, modeling bespoke neighborhoods and prototypical neighborhood types, and modeling neighborhood change using classic and spatial statistical methods. It also provides a set of static and interactive visualization tools to help you display and understand the critical information at each step of the process.

## [mesa-geo](#)

[mesa-geo](#) implements a GeoSpace that can host GIS-based GeoAgents, which are like normal Agents, except they have a shape attribute that is a [Shapely](#) object. You can use [Shapely](#) directly to create arbitrary shapes, but in most cases you will want to import your shapes from a file. Mesa-geo allows you to create GeoAgents from any vector data file (e.g. shapefiles), valid GeoJSON objects or a GeoPandas [GeoDataFrame](#).

## [Pyspatialml](#)

[Pyspatialml](#) is a Python module for applying [scikit-learn](#) machine learning models to ‘stacks’ of raster datasets. Pyspatialml includes functions and classes for working with multiple raster datasets and performing a typical machine learning workflow consisting of extracting training data and applying the predict or [predict\\_proba](#) methods of [scikit-learn](#) estimators to a stack of raster datasets. Pyspatialml

is built upon the `rasterio` Python module for all of the heavy lifting, and is also designed for working with vector data using the `geopandas` module.

## PyGMI

`PyGMI` stands for Python Geoscience Modelling and Interpretation. It is a modelling and interpretation suite aimed at magnetic, gravity and other datasets.

## Visualization

### hvPlot

`hvPlot` provides interactive Bokeh-based plotting for GeoPandas dataframes and series using the same API as the Matplotlib `.plot()` support that comes with GeoPandas. hvPlot makes it simple to pan and zoom into your plots, use widgets to explore multidimensional data, and render even the largest datasets in web browsers using [Datashader](#).

### contextily

`contextily` is a small Python 3 (3.6 and above) package to retrieve tile maps from the internet. It can add those tiles as basemap to `matplotlib` figures or write tile maps to disk into geospatial raster files. Bounding boxes can be passed in both WGS84 (EPSG:4326) and Spheric Mercator (EPSG:3857).

### cartopy

`cartopy` is a Python package designed to make drawing maps for data analysis and visualisation easy. It features: object oriented projection definitions; point, line, polygon and image transformations between projections; integration to expose advanced mapping in `Matplotlib` with a simple and intuitive interface; powerful vector data handling by integrating shapefile reading with `Shapely` capabilities.

### bokeh

`Bokeh` is an interactive visualization library for modern web browsers. It provides elegant, concise construction of versatile graphics, and affords high-performance interactivity over large or streaming datasets. `Bokeh` can help anyone who would like to quickly and easily make interactive plots, dashboards, and data applications.

### folium

`folium` builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the `Leaflet.js` library. Manipulate your data in Python, then visualize it in a `Leaflet` map via `folium`.

## kepler.gl

`Kepler.gl` is a data-agnostic, high-performance web-based application for visual exploration of large-scale geolocation data sets. Built on top of Mapbox GL and `deck.gl`, `kepler.gl` can render millions of points representing thousands of trips and perform spatial aggregations on the fly.

## geoplot

`geoplot` is a high-level Python geospatial plotting library. It's an extension to `cartopy` and `matplotlib` which makes mapping easy: like `seaborn` for geospatial. It comes with the high-level plotting API, native projection support and compatibility with `matplotlib`.

## GeoViews

`GeoViews` is a Python library that makes it easy to explore and visualize any data that includes geographic locations, with native support for GeoPandas dataframes and series objects. It has particularly powerful support for multidimensional meteorological and oceanographic datasets, such as those used in weather, climate, and remote sensing research, but is useful for almost anything that you would want to plot on a map!

## EarthPy

`EarthPy` is a python package that makes it easier to plot and work with spatial raster and vector data using open source tools. `EarthPy` depends upon `geopandas` which has a focus on vector data and `rasterio` with facilitates input and output of raster data files. It also requires `matplotlib` for plotting operations. `EarthPy's` goal is to make working with spatial data easier for scientists.

## splot

`splot` provides statistical visualizations for spatial analysis. It methods for visualizing global and local spatial autocorrelation (through Moran scatterplots and cluster maps), temporal analysis of cluster dynamics (through heatmaps and rose diagrams), and multivariate choropleth mapping (through value-by-alpha maps). A high level API supports the creation of publication-ready visualizations

## legendgram

[legendgram](#) is a small package that provides “legendgrams” legends that visualize the distribution of observations by color in a given map. These distributional visualizations for map classification schemes assist in analytical cartography and spatial data visualization.

## [buckaroo](#)

[buckaroo](#) is a modern data table for Jupyter that expedites the most common exploratory data analysis tasks. It provides scrollable tables, histograms, and summary stats. Buckaroo supports many DataFrame libraries including [geopandas](#). It can display [GeoDataFrame](#)s as tables, it also supports rendering the Geometry as an SVG in the table.

## Geometry manipulation

### [TopoJSON](#)

[topojson](#) is a library for creating a TopoJSON encoding of nearly any geographical object in Python. With topojson it is possible to reduce the size of your geographical data, typically by orders of magnitude. It is able to do so through eliminating redundancy through computation of a topology, fixed-precision integer encoding of coordinates, and simplification and quantization of arcs.

### [geocube](#)

Tool to convert geopandas vector data into rasterized [xarray](#) data.

## Data retrieval

### [OSMnx](#)

[OSMnx](#) is a Python package that lets you download spatial data from OpenStreetMap and model, project, visualize, and analyze real-world street networks. You can download and model walkable, drivable, or bikeable urban networks with a single line of Python code and then easily analyze and visualize them. You can just as easily download and work with other infrastructure types, amenities/points of interest, building footprints, elevation data, street bearings/orientations, and speed/travel time.

### [pyrosm](#)

[Pyrosm](#) is a Python library for reading OpenStreetMap data from Protocolbuffer Binary Format -files (`*.osm.pbf`) into Geopandas [GeoDataFrames](#). [Pyrosm](#) makes it easy to extract various datasets from OpenStreetMap pbf-dumps including e.g. road networks, buildings, Points of Interest (POI), landuse and

natural elements. Also fully customized queries are supported which makes it possible to parse the data from OSM with more specific filters.

## [geobr](#)

[geobr](#) is a computational package to download official spatial data sets of Brazil. The package includes a wide range of geospatial data in geopackage format (like shapefiles but better), available at various geographic scales and for various years with harmonized attributes, projection and topology.

## [cenpy](#)

An interface to explore and query the US Census API and return Pandas [Dataframes](#). This package is intended for exploratory data analysis and draws inspiration from sqlalchemy-like interfaces and [acs.R](#). With separate APIs for application developers and folks who only want to get their data quickly & painlessly, [cenpy](#) should meet the needs of most who aim to get US Census Data into Python.

## [pygadm](#)

[pygadm](#) is a Python package that lets you request spatial data from [GADM](#) without manually downloading any file. This package aims at simplifying the requests of the data using few parameters such as the name

© Copyright 2013–, GeoPandas developers.

Built with the [PyData Sphinx Theme](#) 0.15.4.

Created using [Sphinx](#) 7.3.7.

## Community

---

GeoPandas is a community-led project written, used and supported by a wide range of people from all around the world of a large variety of backgrounds. Everyone is welcome, each small contribution, no matter if it is a fix of a typo in the documentation, bug report, an idea, or a question, is valuable. As a member of our community, you should adhere to the principles presented in the :doc:`Code of Conduct <community/code\_of\_conduct>`.

If you'd like to contribute, please read the :doc:`Contributing guide <community/contributing>`. It will help you to understand the way GeoPandas development works and us to review your contribution.

GeoPandas is a part of the broader Python :doc:`ecosystem <community/ecosystem>`. It depends on a range of great tools, and various packages are built on top of GeoPandas addressing specific needs in geospatial data processing, analysis and visualization.

```
.. container:: button
```

```
:doc:`Contributing <community/contributing>` :doc:`Code of Conduct <community/code_of_conduct>`
:doc:`Ecosystem <community/ecosystem>`
```

```
.. toctree::
```

```
:maxdepth: 2
```

```
:caption: Community
```

```
:hidden:
```

```
Contributing <community/contributing>
```

```
Code of Conduct <community/code_of_conduct>
```

```
Ecosystem <community/ecosystem>
```

# Installation

GeoPandas depends for its spatial functionality on a large geospatial, open source stack of libraries ([GEOS](#), [GDAL](#), [PROJ](#)). See the [Dependencies](#) section below for more details. Those base C libraries can sometimes be a challenge to install. Therefore, we advise you to closely follow the recommendations below to avoid installation problems.

## Installing with Anaconda / conda

To install GeoPandas and all its dependencies, we recommend to use the [conda](#) package manager. This can be obtained by installing the [Anaconda Distribution](#) (a free Python distribution for data science), or through [miniconda](#) (minimal distribution only containing Python and the [conda](#) package manager). See also the [Conda installation docs](#) for more information on how to install Anaconda or miniconda locally.

The advantage of using the [conda](#) package manager is that it provides pre-built binaries for all the required and optional dependencies of GeoPandas for all platforms (Windows, Mac, Linux).

To install the latest version of GeoPandas, you can then do:

```
conda install geopandas
```

## Using the conda-forge channel

[conda-forge](#) is a community effort that provides conda packages for a wide range of software. It provides the *conda-forge* package channel for conda from which packages can be installed, in addition to the “*defaults*” channel provided by Anaconda. Depending on what other packages you are working with, the *defaults* channel or *conda-forge* channel may be better for your needs (e.g. some packages are available on *conda-forge* and not on *defaults*). Generally, *conda-forge* is more up to date with the latest versions of geospatial python packages.

GeoPandas and all its dependencies are available on the *conda-forge* channel, and can be installed as:

```
conda install --channel conda-forge geopandas
```

## Note

We strongly recommend to either install everything from the *defaults* channel, or everything from the *conda-forge* channel. Ending up with a mixture of packages from both channels for the dependencies of GeoPandas can lead to import problems. See the [conda-forge section on using multiple channels](#) for more details.

## Creating a new environment

Creating a new environment is not strictly necessary, but given that installing other geospatial packages from different channels may cause dependency conflicts (as mentioned in the note above), it can be good practice to install the geospatial stack in a clean environment starting fresh.

The following commands create a new environment with the name `geo_env`, configures it to install packages always from conda-forge, and installs GeoPandas in it:

```
conda create -n geo_env
conda activate geo_env
conda config --env --add channels conda-forge
conda config --env --set channel_priority strict
conda install python=3 geopandas
```

## Installing with pip

GeoPandas can also be installed with pip, if all dependencies can be installed as well:

```
pip install geopandas
```

### Warning

When using pip to install GeoPandas, you need to make sure that all dependencies are installed correctly.

Our main dependencies ([shapely](#), [pyproj](#), [pyogrio](#)) provide binary wheels with dependencies included for Mac, Linux, and Windows.

However, depending on your platform or Python version, there might be no pre-compiled wheels available, and then you need to compile and install their C dependencies manually. We refer to the individual packages for more details on installing those. Using conda (see above) avoids the need to compile the dependencies yourself.

Optional runtime dependencies can also be installed all at once:

```
pip install 'geopandas[all]'
```

# Installing from source

You may install the latest development version by cloning the *GitHub* repository and using pip to install from the local directory:

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

Development dependencies can be installed using the dev optional dependency group:

```
pip install '.[dev]'
```

It is also possible to install the latest development version directly from the GitHub repository with:

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

For installing GeoPandas from source, the same [note](#) on the need to have all dependencies correctly installed applies. But, those dependencies can also be installed independently with conda before installing GeoPandas from source:

```
conda install pandas pyogrio shapely pyproj
```

See the [section on conda](#) above for more details on getting running with Anaconda.

# Dependencies

Required dependencies:

- [numpy](#)
- [pandas](#) (version 1.4 or later)
- [shapely](#) (interface to [GEOS](#); version 2.0.0 or later)
- [pyogrio](#) (interface to [GDAL](#); version 0.7.2 or later)
- [pyproj](#) (interface to [PROJ](#); version 3.3.0 or later)
- [packaging](#)

Further, optional dependencies are:

- [fiona](#) (optional; slower alternative to *pyogrio*)
- [psycopg](#) (optional; for PostGIS connection)
- [psycopg2](#) (optional; for PostGIS connection - older version of *psycopg* library)
- [GeoAlchemy2](#) (optional; for writing to PostGIS)
- [geopy](#) (optional; for geocoding)
- [pointpats](#) (optional; for advanced point sampling)

For plotting, these additional packages may be used:

---

© Copyright 2013–, GeoPandas developers.

Created using [Sphinx](#) 7.3.7.

Built with the [PyData Sphinx Theme](#) 0.15.4.