pygeos

Casper van der Wel

Dec 12, 2022

CONTENTS:

1 What is a ufunc? 3

2 Multithreading 5

3 Examples 7

4 References 9

5 Copyright & License 11

Python Module Index 105

Index 107

PyGEOS is a C/Python library with vectorized geometry functions. The geometry operations are done in the open-source geometry library GEOS. PyGEOS wraps these operations in NumPy ufuncs providing a performance improvement when operating on arrays of geometries.

Important note: PyGEOS was merged with Shapely (https://shapely.readthedocs.io) in December 2021 and will be released as part of Shapely 2.0. The development will take place at the Shapely repository. Please raise issues or create pull request over there. PyGEOS itself will receive updates (by backporting from the Shapely repository) until Shapely 2.0 is actually released.

CHAPTER

ONE

WHAT IS A UFUNC?

A universal function (or ufunc for short) is a function that operates on n-dimensional arrays in an element-by-element fashion, supporting array broadcasting. The for-loops that are involved are fully implemented in C diminishing the overhead of the Python interpreter.

CHAPTER

TWO

MULTITHREADING

PyGEOS functions support multithreading. More specifically, the Global Interpreter Lock (GIL) is released during function execution. Normally in Python, the GIL prevents multiple threads from computing at the same time. PyGEOS functions internally releases this constraint so that the heavy lifting done by GEOS can be done in parallel, from a single Python process.

CHAPTER

THREE

EXAMPLES

Compare an grid of points with a polygon:

```
>>> geoms = points(*np.indices((4, 4)))
>>> polygon = box(0, 0, 2, 2)

>>> contains(polygon, geoms)

  array([[False, False, False, False],
         [False,  True, False, False],
         [False, False, False, False],
         [False, False, False, False]])
```

Compute the area of all possible intersections of two lists of polygons:

```
>>> from pygeos import box, area, intersection

>>> polygons_x = box(range(5), 0, range(10, 15), 10)
>>> polygons_y = box(0, range(5), 10, range(10, 15))

>>> area(intersection(polygons_x[:, np.newaxis], polygons_y[np.newaxis, :]))

array([[100.,  90.,  80.,  70.,  60.],
       [ 90.,  81.,  72.,  63.,  54.],
       [ 80.,  72.,  64.,  56.,  48.],
       [ 70.,  63.,  56.,  49.,  42.],
       [ 60.,  54.,  48.,  42.,  36.]])
```

See the documentation for more: https://pygeos.readthedocs.io

CHAPTER

FOUR

REFERENCES

- GEOS: https://libgeos.org
- Shapely: https://shapely.readthedocs.io/en/latest/
- Numpy ufuncs: https://docs.scipy.org/doc/numpy/reference/ufuncs.html
- Joris van den Bossche's blogpost: https://jorisvandenbossche.github.io/blog/2017/09/19/geopandas-cython/
- Matthew Rocklin's blogpost: http://matthewrocklin.com/blog/work/2017/09/21/accelerating-geopandas-1

CHAPTER

FIVE

COPYRIGHT & LICENSE

PyGEOS is licensed under BSD 3-Clause license. Copyright (c) 2019, Casper van der Wel. GEOS is available under the terms of GNU Lesser General Public License (LGPL) 2.1 at https://libgeos.org.

5.1 API Reference

5.1.1 Installation

Installation from PyPI

PyGEOS is available as a binary distribution (wheel) for Linux, OSX and Windows platforms. The distribution includes a GEOS version that was most recent at the time of the PyGEOS release. Install the binary wheel with pip as follows:

```
$ pip install pygeos
```

Installation using Anaconda

PyGEOS is available on the conda-forge channel. Install as follows:

```
$ conda install pygeos --channel conda-forge
```

Installation with custom GEOS libary

You may want to use a specific GEOS version or a GEOS distribution that is already present on your system. In such cases you will need to compile PyGEOS yourself.

On Linux:

```
$ sudo apt install libgeos-dev  # skip this if you already have GEOS
$ pip install pygeos --no-binary
```

On OSX:

```
$ brew install geos  # skip this if you already have GEOS
$ pip install pygeos --no-binary
```

We do not have a recipe for Windows platforms. The following steps should enable you to build PyGEOS yourself:

- Get a C compiler applicable to your Python version (https://wiki.python.org/moin/WindowsCompilers)

- Download and install a GEOS binary (https://trac.osgeo.org/osgeo4w/)
- Set GEOS_INCLUDE_PATH and GEOS_LIBRARY_PATH environment variables (see below for notes on GEOS discovery)
- Run `pip install pygeos --no-binary`
- Make sure the GEOS .dll files are available on the PATH

Installation from source

The same as installation with a custom GEOS binary, but then instead of installing pygeos with pip, you clone the package from Github:

```
$ git clone git@github.com:pygeos/pygeos.git
```

Install it in development mode using `pip`:

```
$ pip install -e .[test]
```

Testing PyGEOS

PyGEOS can be tested using `pytest`:

```
$ pip install pytest  # or pygeos[test]
$ pytest --pyargs pygeos.tests
```

GEOS discovery (compile time)

If GEOS is installed on Linux or OSX, normally the `geos-config` command line utility will be available and `pip` will find GEOS automatically. If the correct `geos-config` is not on the PATH, you can add it as follows (on Linux/OSX):

```
$ export PATH=/path/to/geos/bin:$PATH
```

Alternatively, you can specify where PyGEOS should look for GEOS (on Linux/OSX):

```
$ export GEOS_INCLUDE_PATH=/path/to/geos/include
$ export GEOS_LIBRARY_PATH=/path/to/geos/lib
```

On Windows, there is no `geos-config` and the include and lib folders need to be specified manually in any case:

```
$ set GEOS_INCLUDE_PATH=C:\path\to\geos\include
$ set GEOS_LIBRARY_PATH=C:\path\to\geos\lib
```

Common locations of GEOS (to be suffixed by `lib`, `include` or `bin`):

- Anaconda (Linux/OSX): `$CONDA_PREFIX/Library`
- Anaconda (Windows): `%CONDA_PREFIX%\Library`
- OSGeo4W (Windows): `C:\OSGeo4W64`

GEOS discovery (runtime)

PyGEOS is dynamically linked to GEOS. This means that the same GEOS library that was used during PyGEOS compilation is required on your system at runtime. When using pygeos that was distributed as a binary wheel or through conda, this is automatically the case and you can stop reading.

In other cases this can be tricky, especially if you have multiple GEOS installations next to each other. We only include some guidelines here to address this issue as this document is not intended as a general guide of shared library discovery.

If you encounter exceptions like:

```
ImportError: libgeos_c.so.1: cannot open shared object file: No such file or directory
```

You will have to make the shared library file available to the Python interpreter. There are in general four ways of making Python aware of the location of shared library:

1. Copy the shared libraries into the pygeos module directory (this is how Windows binary wheels work: they are distributed with the correct dlls in the pygeos module directory)
2. Copy the shared libraries into the library directory of the Python interpreter (this is how Anaconda environments work)
3. Copy the shared libraries into some system location (`C:\Windows\System32`; `/usr/local/lib`, this happens if you installed GEOS through `apt` or `brew`)
4. Add the shared library location to a the dynamic linker path variable at runtime. (Advanced usage; Linux and OSX only; on Windows this method was deprecated in Python 3.8)

The filenames of the GEOS shared libraries are:

- On Linux: `libgeos-*.so.*, libgeos_c-*.so.*`
- On OSX: `libgeos.dylib, libgeos_c.dylib`
- On Windows: `geos-*.dll, geos_c-*.dll`

Note that pygeos does not make use of any RUNPATH (RPATH) header. The location of the GEOS shared library is not stored inside the compiled PyGEOS library.

5.1.2 Geometry

The `pygeos.Geometry` class is the central datatype in PyGEOS. An instance of `Geometry` is a container of the actual GEOSGeometry object. The Geometry object keeps track of the underlying GEOSGeometry and lets the python garbage collector free its memory when it is not used anymore.

Geometry objects are immutable. This means that after constructed, they cannot be changed in place. Every PyGEOS operation will result in a new object being returned.

Construction

For convenience, the `Geometry` class can be constructed with a WKT (Well-Known Text) or WKB (Well-Known Binary) representation of a geometry:

```
>>> from pygeos import Geometry
>>> point_1 = Geometry("POINT (5.2 52.1)")
>>> point_2 = Geometry(b"\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\
↪x00\x00\x00\xf0?")
```

A more efficient way of constructing geometries is by making use of the (vectorized) functions described in *pygeos.creation*.

Pickling

Geometries can be serialized using pickle:

```
>>> import pickle
>>> pickled = pickle.dumps(point_1)
>>> pickle.loads(point_1)
<pygeos.Geometry POINT (5.2 52.1)>
```

Warning: Pickling will convert linearrings to linestrings. See *pygeos.io.to_wkb()* for a complete list of limitations.

Hashing

Geometries can be used as elements in sets or as keys in dictionaries. Python uses a technique called *hashing* for lookups in these datastructures. PyGEOS generates this hash from the WKB representation. Therefore, geometries are equal if and only if their WKB representations are equal.

```
>>> point_3 = Geometry("POINT (5.2 52.1)")
>>> {point_1, point_2, point_3}
{<pygeos.Geometry POINT (5.2 52.1)>, <pygeos.Geometry POINT (1 1)>}
```

Warning: Due to limitations of WKB, linearrings will equal linestrings if they contain the exact same points. See *pygeos.io.to_wkb()*.

Comparing two geometries directly is also supported. This is the same as using *pygeos.predicates.equals_exact()* with a `tolerance` value of zero.

```
>>> point_1 == point_2
False
>>> point_1 != point_2
True
```

Properties

Geometry objects have neither properties nor methods. Instead, use the functions listed below to obtain information about geometry objects.

force_2d(*geometry*, ***kwargs*)

Forces the dimensionality of a geometry to 2D.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> force_2d(Geometry("POINT Z (0 0 0)"))
<pygeos.Geometry POINT (0 0)>
>>> force_2d(Geometry("POINT (0 0)"))
<pygeos.Geometry POINT (0 0)>
>>> force_2d(Geometry("LINESTRING (0 0 0, 0 1 1, 1 1 2)"))
<pygeos.Geometry LINESTRING (0 0, 0 1, 1 1)>
>>> force_2d(Geometry("POLYGON Z EMPTY"))
<pygeos.Geometry POLYGON EMPTY>
>>> force_2d(None) is None
True
```

force_3d(*geometry*, *z=0.0*, ***kwargs*)

Forces the dimensionality of a geometry to 3D.

2D geometries will get the provided Z coordinate; Z coordinates of 3D geometries are unchanged (unless they are nan).

Note that for empty geometries, 3D is only supported since GEOS 3.9 and then still only for simple geometries (non-collections).

Parameters

geometry
[Geometry or array_like]

z
[float or array_like, default 0.0]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> force_3d(Geometry("POINT (0 0)"), z=3)
<pygeos.Geometry POINT Z (0 0 3)>
>>> force_3d(Geometry("POINT Z (0 0 0)"), z=3)
<pygeos.Geometry POINT Z (0 0 0)>
>>> force_3d(Geometry("LINESTRING (0 0, 0 1, 1 1)"))
<pygeos.Geometry LINESTRING Z (0 0 0, 0 1 0, 1 1 0)>
>>> force_3d(None) is None
True
```

get_coordinate_dimension(*geometry*, ***kwargs*)

Returns the dimensionality of the coordinates in a geometry (2 or 3).

Returns -1 for missing geometries (None values). Note that if the first Z coordinate equals nan, this function will return 2.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> get_coordinate_dimension(Geometry("POINT (0 0)"))
2
>>> get_coordinate_dimension(Geometry("POINT Z (0 0 0)"))
3
>>> get_coordinate_dimension(None)
-1
>>> get_coordinate_dimension(Geometry("POINT Z (0 0 nan)"))
2
```

get_dimensions(*geometry*, ***kwargs*)

Returns the inherent dimensionality of a geometry.

The inherent dimension is 0 for points, 1 for linestrings and linearrings, and 2 for polygons. For geometrycollections it is the max of the containing elements. Empty collections and None values return -1.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> get_dimensions(Geometry("POINT (0 0)"))
0
>>> get_dimensions(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
2
>>> get_dimensions(Geometry("GEOMETRYCOLLECTION (POINT(0 0), LINESTRING(0 0, 1 1))
↪"))
1
>>> get_dimensions(Geometry("GEOMETRYCOLLECTION EMPTY"))
-1
>>> get_dimensions(None)
-1
```

get_exterior_ring(*geometry*, ***kwargs*)

Returns the exterior ring of a polygon.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_interior_ring

Examples

```
>>> get_exterior_ring(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
<pygeos.Geometry LINEARRING (0 0, 0 10, 10 10, 10 0, 0 0)>
>>> get_exterior_ring(Geometry("POINT (1 1)")) is None
True
```

get_geometry(*geometry*, *index*, ***kwargs*)

Returns the nth geometry from a collection of geometries.

Parameters

geometry
[Geometry or array_like]

index
[int or array_like] Negative values count from the end of the collection backwards.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_num_geometries, *get_parts*

Notes

- simple geometries act as length-1 collections
- out-of-range values return None

Examples

```
>>> multipoint = Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)")
>>> get_geometry(multipoint, 1)
<pygeos.Geometry POINT (1 1)>
>>> get_geometry(multipoint, -1)
<pygeos.Geometry POINT (3 3)>
>>> get_geometry(multipoint, 5) is None
True
>>> get_geometry(Geometry("POINT (1 1)"), 0)
<pygeos.Geometry POINT (1 1)>
>>> get_geometry(Geometry("POINT (1 1)"), 1) is None
True
```

get_interior_ring(*geometry*, *index*, ***kwargs*)

Returns the nth interior ring of a polygon.

Parameters

geometry
[Geometry or array_like]

index
[int or array_like] Negative values count from the end of the interior rings backwards.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_exterior_ring
get_num_interior_rings

Examples

```
>>> polygon_with_hole = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2 4,␣
↪4 4, 4 2, 2 2))")
>>> get_interior_ring(polygon_with_hole, 0)
<pygeos.Geometry LINEARRING (2 2, 2 4, 4 4, 4 2, 2 2)>
>>> get_interior_ring(Geometry("POINT (1 1)"), 0) is None
True
```

get_num_coordinates(*geometry*, ***kwargs*)

Returns the total number of coordinates in a geometry.

Returns 0 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> get_num_coordinates(Geometry("POINT (0 0)"))
1
>>> get_num_coordinates(Geometry("POINT Z (0 0 0)"))
1
>>> get_num_coordinates(Geometry("GEOMETRYCOLLECTION (POINT(0 0), LINESTRING(0 0, 1␣
↪1))"))
3
>>> get_num_coordinates(None)
0
```

get_num_geometries(*geometry*, ***kwargs*)

Returns number of geometries in a collection.

Returns 0 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like] The number of geometries in points, linestrings, linearrings and polygons equals one.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_num_points
get_geometry

Examples

```
>>> get_num_geometries(Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)"))
4
>>> get_num_geometries(Geometry("POINT (1 1)"))
1
>>> get_num_geometries(None)
0
```

get_num_interior_rings(*geometry*, ***kwargs*)

Returns number of internal rings in a polygon

Returns 0 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like] The number of interior rings in non-polygons equals zero.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_exterior_ring
get_interior_ring

Examples

```
>>> polygon = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))")
>>> get_num_interior_rings(polygon)
0
>>> polygon_with_hole = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2 4,␣
↪4 4, 4 2, 2 2))")
>>> get_num_interior_rings(polygon_with_hole)
1
>>> get_num_interior_rings(Geometry("POINT (1 1)"))
0
>>> get_num_interior_rings(None)
0
```

get_num_points(*geometry*, ***kwargs*)

Returns number of points in a linestring or linearring.

Returns 0 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like] The number of points in geometries other than linestring or linearring equals zero.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_point
get_num_geometries

Examples

```
>>> line = Geometry("LINESTRING (0 0, 1 1, 2 2, 3 3)")
>>> get_num_points(line)
4
>>> get_num_points(Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)"))
0
>>> get_num_points(None)
0
```

get_parts(*geometry*, *return_index=False*)

Gets parts of each GeometryCollection or Multi* geometry object; returns a copy of each geometry in the GeometryCollection or Multi* geometry object.

Note: This does not return the individual parts of Multi* geometry objects in a GeometryCollection. You may need to call this function multiple times to return individual parts of Multi* geometry objects in a GeometryCollection.

Parameters

geometry
[Geometry or array_like]

return_index
[bool, default False] If True, will return a tuple of ndarrays of (parts, indexes), where indexes are the indexes of the original geometries in the source array.

Returns

ndarray of parts or tuple of (parts, indexes)

See also:

get_geometry, *get_rings*

Examples

```
>>> get_parts(Geometry("MULTIPOINT (0 1, 2 3)")).tolist()
[<pygeos.Geometry POINT (0 1)>, <pygeos.Geometry POINT (2 3)>]
>>> parts, index = get_parts([Geometry("MULTIPOINT (0 1)"), Geometry("MULTIPOINT (4␣
↪5, 6 7)")], return_index=True)
>>> parts.tolist()
[<pygeos.Geometry POINT (0 1)>, <pygeos.Geometry POINT (4 5)>, <pygeos.Geometry␣
↪POINT (6 7)>]
>>> index.tolist()
[0, 1, 1]
```

get_point(*geometry*, *index*, ***kwargs*)

Returns the nth point of a linestring or linearring.

Parameters

geometry
[Geometry or array_like]

index
[int or array_like] Negative values count from the end of the linestring backwards.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_num_points

Examples

```
>>> line = Geometry("LINESTRING (0 0, 1 1, 2 2, 3 3)")
>>> get_point(line, 1)
<pygeos.Geometry POINT (1 1)>
>>> get_point(line, -2)
<pygeos.Geometry POINT (2 2)>
>>> get_point(line, [0, 3]).tolist()
[<pygeos.Geometry POINT (0 0)>, <pygeos.Geometry POINT (3 3)>]
>>> get_point(Geometry("LINEARRING (0 0, 1 1, 2 2, 0 0)"), 1)
<pygeos.Geometry POINT (1 1)>
>>> get_point(Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)"), 1) is None
True
>>> get_point(Geometry("POINT (1 1)"), 0) is None
True
```

get_precision(*geometry*, ***kwargs*)

Get the precision of a geometry.

Note: 'get_precision' requires at least GEOS 3.6.0.

If a precision has not been previously set, it will be 0 (double precision). Otherwise, it will return the precision grid size that was set on a geometry.

Returns NaN for not-a-geometry values.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

set_precision

Examples

```
>>> get_precision(Geometry("POINT (1 1)"))
0.0
>>> geometry = set_precision(Geometry("POINT (1 1)"), 1.0)
>>> get_precision(geometry)
1.0
>>> np.isnan(get_precision(None))
True
```

get_rings(*geometry*, *return_index=False*)

Gets rings of Polygon geometry object.

For each Polygon, the first returned ring is always the exterior ring and potential subsequent rings are interior rings.

If the geometry is not a Polygon, nothing is returned (empty array for scalar geometry input or no element in output array for array input).

Parameters

geometry
[Geometry or array_like]

return_index
[bool, default False] If True, will return a tuple of ndarrays of (rings, indexes), where indexes are the indexes of the original geometries in the source array.

Returns

ndarray of rings or tuple of (rings, indexes)

See also:

get_exterior_ring, *get_interior_ring*, *get_parts*

Examples

```
>>> polygon_with_hole = Geometry("POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2 4,
↪ 4 4, 4 2, 2 2))")
>>> get_rings(polygon_with_hole).tolist()
[<pygeos.Geometry LINEARRING (0 0, 0 10, 10 10, 10 0, 0 0)>,
 <pygeos.Geometry LINEARRING (2 2, 2 4, 4 4, 4 2, 2 2)>]
```

With `return_index=True`:

```
>>> polygon = Geometry("POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))")
>>> rings, index = get_rings([polygon, polygon_with_hole], return_index=True)
>>> rings.tolist()
[<pygeos.Geometry LINEARRING (0 0, 2 0, 2 2, 0 2, 0 0)>,
 <pygeos.Geometry LINEARRING (0 0, 0 10, 10 10, 10 0, 0 0)>,
 <pygeos.Geometry LINEARRING (2 2, 2 4, 4 4, 4 2, 2 2)>]
>>> index.tolist()
[0, 1, 1]
```

get_srid(*geometry*, ***kwargs*)

Returns the SRID of a geometry.

Returns -1 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

set_srid

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> with_srid = set_srid(point, 4326)
>>> get_srid(point)
0
>>> get_srid(with_srid)
4326
```

get_type_id(*geometry*, ***kwargs*)

Returns the type ID of a geometry.

- None (missing) is -1
- POINT is 0
- LINESTRING is 1
- LINEARRING is 2
- POLYGON is 3

- MULTIPOINT is 4
- MULTILINESTRING is 5
- MULTIPOLYGON is 6
- GEOMETRYCOLLECTION is 7

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

`GeometryType`

Examples

```
>>> get_type_id(Geometry("LINESTRING (0 0, 1 1, 2 2, 3 3)"))
1
>>> get_type_id([Geometry("POINT (1 2)"), Geometry("POINT (1 2)")]).tolist()
[0, 0]
```

`get_x`(*point*, ***kwargs*)

Returns the x-coordinate of a point

Parameters

point
[Geometry or array_like] Non-point geometries will result in NaN being returned.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

`get_y`, *`get_z`*

Examples

```
>>> get_x(Geometry("POINT (1 2)"))
1.0
>>> get_x(Geometry("MULTIPOINT (1 1, 1 2)"))
nan
```

`get_y`(*point*, ***kwargs*)

Returns the y-coordinate of a point

Parameters

point
[Geometry or array_like] Non-point geometries will result in NaN being returned.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_x, *get_z*

Examples

```
>>> get_y(Geometry("POINT (1 2)"))
2.0
>>> get_y(Geometry("MULTIPOINT (1 1, 1 2)"))
nan
```

get_z(*point*, ***kwargs*)

Returns the z-coordinate of a point.

Note: 'get_z' requires at least GEOS 3.7.0.

Parameters

point
[Geometry or array_like] Non-point geometries or geometries without 3rd dimension will result in NaN being returned.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_x, *get_y*

Examples

```
>>> get_z(Geometry("POINT Z (1 2 3)"))
3.0
>>> get_z(Geometry("POINT (1 2)"))
nan
>>> get_z(Geometry("MULTIPOINT Z (1 1 1, 2 2 2)"))
nan
```

set_precision(*geometry*, *grid_size*, *mode='valid_output'*, ***kwargs*)

Returns geometry with the precision set to a precision grid size.

Note: 'set_precision' requires at least GEOS 3.6.0.

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. Z values, if present, will not be modified.

Note: 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.

Also note: input geometries should be geometrically valid; unexpected results may occur if input geometries are not.

Returns None if geometry is None.

Parameters

geometry
[Geometry or array_like]

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 how to handle invalid output geometries. There are three modes:

1. *'valid_output'* (default): The output is always valid. Collapsed geometry elements (including both polygons and lines) are removed. Duplicate vertices are removed.
2. *'pointwise'*: Precision reduction is performed pointwise. Output geometry may be invalid due to collapse or self-intersection. Duplicate vertices are not removed. In GEOS this option is called NO_TOPO.

   Note: 'pointwise' mode requires at least GEOS 3.10. It is accepted in earlier versions, but the results may be unexpected.

   ---

3. *'keep_collapsed'*: Like the default mode, except that collapsed linear geometry elements are preserved. Collapsed polygonal input elements are removed. Duplicate vertices are removed.

preserve_topology
[bool, optional] Deprecated since version 0.11: This parameter is ignored. Use `mode` instead.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_precision

Examples

```
>>> set_precision(Geometry("POINT (0.9 0.9)"), 1.0)
<pygeos.Geometry POINT (1 1)>
>>> set_precision(Geometry("POINT (0.9 0.9 0.9)"), 1.0)
<pygeos.Geometry POINT Z (1 1 0.9)>
>>> set_precision(Geometry("LINESTRING (0 0, 0 0.1, 0 1, 1 1)"), 1.0)
<pygeos.Geometry LINESTRING (0 0, 0 1, 1 1)>
>>> set_precision(Geometry("LINESTRING (0 0, 0 0.1, 0.1 0.1)"), 1.0, mode="valid_
↪output")
<pygeos.Geometry LINESTRING Z EMPTY>
```

(continues on next page)

(continued from previous page)

```
>>> set_precision(Geometry("LINESTRING (0 0, 0 0.1, 0.1 0.1)"), 1.0, mode="pointwise
↪")
<pygeos.Geometry LINESTRING (0 0, 0 0, 0 0)>
>>> set_precision(Geometry("LINESTRING (0 0, 0 0.1, 0.1 0.1)"), 1.0, mode="keep_
↪collapsed")
<pygeos.Geometry LINESTRING (0 0, 0 0)>
>>> set_precision(None, 1.0) is None
True
```

set_srid(*geometry*, *srid*, ***kwargs*)

Returns a geometry with its SRID set.

Parameters

geometry
[Geometry or array_like]

srid
[int]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_srid

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> with_srid = set_srid(point, 4326)
>>> get_srid(point)
0
>>> get_srid(with_srid)
4326
```

5.1.3 Geometry creation

box(*xmin*, *ymin*, *xmax*, *ymax*, *ccw=True*, ***kwargs*)

Create box polygons.

Parameters

xmin
[array_like]

ymin
[array_like]

xmax
[array_like]

ymax
[array_like]

ccw
[bool, default True] If True, box will be created in counterclockwise direction starting from bottom right coordinate (xmax, ymin). If False, box will be created in clockwise direction starting from bottom left coordinate (xmin, ymin).

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> box(0, 0, 1, 1)
<pygeos.Geometry POLYGON ((1 0, 1 1, 0 1, 0 0, 1 0))>
>>> box(0, 0, 1, 1, ccw=False)
<pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

destroy_prepared(*geometry*, ***kwargs*)

Destroy the prepared part of a geometry, freeing up memory.

Note that the prepared geometry will always be cleaned up if the geometry itself is dereferenced. This function needs only be called in very specific circumstances, such as freeing up memory without losing the geometries, or benchmarking.

Parameters

geometry
[Geometry or array_like] Geometries are changed inplace

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

prepare

empty(*shape*, *geom_type=None*, *order='C'*)

Create a geometry array prefilled with None or with empty geometries.

Parameters

shape
[int or tuple of int] Shape of the empty array, e.g., `(2, 3)` or `2`.

geom_type
[pygeos.GeometryType, optional] The desired geometry type in case the array should be prefilled with empty geometries. Default `None`.

order
[{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

Examples

```
>>> empty((2, 3)).tolist()
[[None, None, None], [None, None, None]]
>>> empty(2, geom_type=GeometryType.POINT).tolist()
[<pygeos.Geometry POINT EMPTY>, <pygeos.Geometry POINT EMPTY>]
```

geometrycollections(*geometries*, *indices=None*, *out=None*, ***kwargs*)

Create geometrycollections from arrays of geometries

Parameters

geometries
[array_like] An array of geometries

indices
[array_like, optional] Indices into the target array where input geometries belong. If provided, both geometries and indices should be 1D and have matching sizes. Indices should be in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
[ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

See also:

multipoints

linearrings(*coords*, *y=None*, *z=None*, *indices=None*, *out=None*, ***kwargs*)

Create an array of linearrings.

If the provided coords do not constitute a closed linestring, or if there are only 3 provided coords, the first coordinate is duplicated at the end to close the ring. This function will raise an exception if a linearring contains less than three points or if the terminal coordinates contain NaN (not-a-number).

Parameters

coords
[array_like] An array of lists of coordinate tuples (2- or 3-dimensional) or, if `y` is provided, an array of lists of x coordinates

y
[array_like, optional]

z
[array_like, optional]

indices
[array_like, optional] Indices into the target array where input coordinates belong. If provided, the coords should be 2D with shape (N, 2) or (N, 3) and indices should be an array of shape (N,) with integers in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
[ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

See also:

linestrings

Notes

- Usage of the `y` and `z` arguments will prevents lazy evaluation in `dask`. Instead provide the coordinates as a `(..., 2)` or `(..., 3)` array using only `coords`.

Examples

```
>>> linearrings([[0, 0], [0, 1], [1, 1], [0, 0]])
<pygeos.Geometry LINEARRING (0 0, 0 1, 1 1, 0 0)>
>>> linearrings([[0, 0], [0, 1], [1, 1]])
<pygeos.Geometry LINEARRING (0 0, 0 1, 1 1, 0 0)>
```

linestrings(*coords*, *y=None*, *z=None*, *indices=None*, *out=None*, ***kwargs*)

Create an array of linestrings.

This function will raise an exception if a linestring contains less than two points.

Parameters

coords
[array_like] An array of lists of coordinate tuples (2- or 3-dimensional) or, if `y` is provided, an array of lists of x coordinates

y
[array_like, optional]

z
[array_like, optional]

indices
[array_like, optional] Indices into the target array where input coordinates belong. If provided, the coords should be 2D with shape (N, 2) or (N, 3) and indices should be an array of shape (N,) with integers in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
[ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

Notes

- Usage of the `y` and `z` arguments will prevents lazy evaluation in `dask`. Instead provide the coordinates as a `(..., 2)` or `(..., 3)` array using only `coords`.

Examples

```
>>> linestrings([[[0, 1], [4, 5]], [[2, 3], [5, 6]]]).tolist()
[<pygeos.Geometry LINESTRING (0 1, 4 5)>, <pygeos.Geometry LINESTRING (2 3, 5 6)>]
>>> linestrings([[0, 1], [4, 5], [2, 3], [5, 6], [7, 8]], indices=[0, 0, 1, 1, 1]).
↪tolist()
[<pygeos.Geometry LINESTRING (0 1, 4 5)>, <pygeos.Geometry LINESTRING (2 3, 5 6, 7␣
↪8)>]
```

multilinestrings(*geometries*, *indices=None*, *out=None*, ***kwargs*)

Create multilinestrings from arrays of linestrings

Parameters

geometries
: [array_like] An array of linestrings or coordinates (see linestrings).

indices
: [array_like, optional] Indices into the target array where input geometries belong. If provided, both geometries and indices should be 1D and have matching sizes. Indices should be in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
: [ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
: For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

See also:

multipoints

multipoints(*geometries*, *indices=None*, *out=None*, ***kwargs*)

Create multipoints from arrays of points

Parameters

geometries
: [array_like] An array of points or coordinates (see points).

indices
: [array_like, optional] Indices into the target array where input geometries belong. If provided, both geometries and indices should be 1D and have matching sizes. Indices should be in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
: [ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

Examples

Multipoints are constructed from points:

```
>>> point_1 = points([1, 1])
>>> point_2 = points([2, 2])
>>> multipoints([point_1, point_2])
<pygeos.Geometry MULTIPOINT (1 1, 2 2)>
>>> multipoints([[point_1, point_2], [point_2, None]]).tolist()
[<pygeos.Geometry MULTIPOINT (1 1, 2 2)>, <pygeos.Geometry MULTIPOINT (2 2)>]
```

Or from coordinates directly:

```
>>> multipoints([[0, 0], [2, 2], [3, 3]])
<pygeos.Geometry MULTIPOINT (0 0, 2 2, 3 3)>
```

Multiple multipoints of different sizes can be constructed efficiently using the `indices` keyword argument:

```
>>> multipoints([point_1, point_2, point_2], indices=[0, 0, 1]).tolist()
[<pygeos.Geometry MULTIPOINT (1 1, 2 2)>, <pygeos.Geometry MULTIPOINT (2 2)>]
```

Missing input values (`None`) are ignored and may result in an empty multipoint:

```
>>> multipoints([None])
<pygeos.Geometry MULTIPOINT EMPTY>
>>> multipoints([point_1, None], indices=[0, 0]).tolist()
[<pygeos.Geometry MULTIPOINT (1 1)>]
>>> multipoints([point_1, None], indices=[0, 1]).tolist()
[<pygeos.Geometry MULTIPOINT (1 1)>, <pygeos.Geometry MULTIPOINT EMPTY>]
```

multipolygons(*geometries*, *indices=None*, *out=None*, ***kwargs*)

Create multipolygons from arrays of polygons

Parameters

geometries
[array_like] An array of polygons or coordinates (see polygons).

indices
[array_like, optional] Indices into the target array where input geometries belong. If provided, both geometries and indices should be 1D and have matching sizes. Indices should be in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
[ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

See also:

multipoints

points(*coords*, *y=None*, *z=None*, *indices=None*, *out=None*, ***kwargs*)

Create an array of points.

Parameters

coords
[array_like] An array of coordinate tuples (2- or 3-dimensional) or, if `y` is provided, an array of x coordinates.

y
[array_like, optional]

z
[array_like, optional]

indices
[array_like, optional] Indices into the target array where input coordinates belong. If provided, the coords should be 2D with shape (N, 2) or (N, 3) and indices should be an array of shape (N,) with integers in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
[ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

Notes

- GEOS >=3.10 automatically converts POINT (nan nan) to POINT EMPTY.
- Usage of the `y` and `z` arguments will prevents lazy evaluation in `dask`. Instead provide the coordinates as an array with shape `(..., 2)` or `(..., 3)` using only the `coords` argument.

Examples

```
>>> points([[0, 1], [4, 5]]).tolist()
[<pygeos.Geometry POINT (0 1)>, <pygeos.Geometry POINT (4 5)>]
>>> points([0, 1, 2])
<pygeos.Geometry POINT Z (0 1 2)>
```

polygons(*geometries*, *holes=None*, *indices=None*, *out=None*, ***kwargs*)

Create an array of polygons.

Parameters

geometries
[array_like] An array of linearrings or coordinates (see linearrings). Unless `indices` are given (see description below), this include the outer shells only. The `holes` argument should be used to create polygons with holes.

holes
[array_like, optional] An array of lists of linearrings that constitute holes for each shell. Not to be used in combination with `indices`.

indices
[array_like, optional] Indices into the target array where input geometries belong. If provided, the holes are expected to be present inside `geometries`; the first geometry for each index is the outer shell and all subsequent geometries in that index are the holes. Both geometries and indices should be 1D and have matching sizes. Indices should be in increasing order. Missing indices result in a ValueError unless `out` is provided, in which case the original value in `out` is kept.

out
[ndarray, optional] An array (with dtype object) to output the geometries into.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs. Ignored if `indices` is provided.

Examples

Polygons are constructed from rings:

```
>>> ring_1 = linearrings([[0, 0], [0, 10], [10, 10], [10, 0]])
>>> ring_2 = linearrings([[2, 6], [2, 7], [3, 7], [3, 6]])
>>> polygons([ring_1, ring_2])[0]
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
>>> polygons([ring_1, ring_2])[1]
<pygeos.Geometry POLYGON ((2 6, 2 7, 3 7, 3 6, 2 6))>
```

Or from coordinates directly:

```
>>> polygons([[0, 0], [0, 10], [10, 10], [10, 0]])
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
```

Adding holes can be done using the `holes` keyword argument:

```
>>> polygons(ring_1, holes=[ring_2])
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0), (2 6, 2 7, 3 7, 3 6...>
```

Or using the `indices` argument:

```
>>> polygons([ring_1, ring_2], indices=[0, 1])[0]
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
>>> polygons([ring_1, ring_2], indices=[0, 1])[1]
<pygeos.Geometry POLYGON ((2 6, 2 7, 3 7, 3 6, 2 6))>
>>> polygons([ring_1, ring_2], indices=[0, 0])[0]
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0), (2 6, 2 7, 3 7, 3 6...>
```

Missing input values (`None`) are ignored and may result in an empty polygon:

```
>>> polygons(None)
<pygeos.Geometry POLYGON EMPTY>
>>> polygons(ring_1, holes=[None])
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
>>> polygons([ring_1, None], indices=[0, 0])[0]
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
```

prepare(*geometry*, ***kwargs*)

Prepare a geometry, improving performance of other operations.

A prepared geometry is a normal geometry with added information such as an index on the line segments. This improves the performance of the following operations: contains, contains_properly, covered_by, covers, crosses, disjoint, intersects, overlaps, touches, and within.

Note that if a prepared geometry is modified, the newly created Geometry object is not prepared. In that case, `prepare` should be called again.

This function does not recompute previously prepared geometries; it is efficient to call this function on an array that partially contains prepared geometries.

Parameters

geometry
[Geometry or array_like] Geometries are changed inplace

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

`is_prepared`
Identify whether a geometry is prepared already.

`destroy_prepared`
Destroy the prepared part of a geometry.

5.1.4 Input/Output

from_geojson(*geometry*, *on_invalid='raise'*, ***kwargs*)

Creates geometries from GeoJSON representations (strings).

Note: 'from_geojson' requires at least GEOS 3.10.1.

If a GeoJSON is a FeatureCollection, it is read as a single geometry (with type GEOMETRYCOLLECTION). This may be unpacked using the `pygeos.get_parts`. Properties are not read.

The GeoJSON format is defined in RFC 7946.

The following are currently unsupported:

- Three-dimensional geometries: the third dimension is ignored.
- Geometries having 'null' in the coordinates.

Parameters

geometry
[str, bytes or array_like] The GeoJSON string or byte object(s) to convert.

on_invalid
[{"raise", "warn", "ignore"}, default "raise"]

- raise: an exception will be raised if an input GeoJSON is invalid.
- warn: a warning will be raised and invalid input geometries will be returned as `None`.
- ignore: invalid input geometries will be returned as `None` without a warning.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

`get_parts`

Examples

```
>>> from_geojson('{"type": "Point","coordinates": [1, 2]}')
<pygeos.Geometry POINT (1 2)>
```

from_shapely(*geometry*, ***kwargs*)

Creates geometries from shapely Geometry objects.

Warning: When Shapely and PyGEOS are using the same GEOS version, this function assumes that the libraries are actually the same. In some cases (especially when using pip-installed wheels) this may lead to unexpected behaviour. If you require safe (but slower) behaviour, then we recommend setting `pygeos.io.shapely_compatible` to `False`.

Parameters

geometry
[shapely Geometry object or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Notes

If PyGEOS and Shapely do not use the same GEOS version, the conversion happens through the WKB format and will thus be slower.

Examples

```
>>> from shapely.geometry import Point
>>> from_shapely(Point(1, 2))
<pygeos.Geometry POINT (1 2)>
```

from_wkb(*geometry*, *on_invalid='raise'*, ***kwargs*)

Creates geometries from the Well-Known Binary (WKB) representation.

The Well-Known Binary format is defined in the OGC Simple Features Specification for SQL.

Parameters

geometry
[str or array_like] The WKB byte object(s) to convert.

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**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> from_wkb(b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\
↪x00\x00\xf0?')
<pygeos.Geometry POINT (1 1)>
```

from_wkt(*geometry*, *on_invalid='raise'*, ***kwargs*)

Creates geometries from the Well-Known Text (WKT) representation.

The Well-known Text format is defined in the OGC Simple Features Specification for SQL.

Parameters

geometry
[str or array_like] The WKT string(s) to convert.

on_invalid
[{"raise", "warn", "ignore"}, default "raise"]

- raise: an exception will be raised if WKT input geometries are 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**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> from_wkt('POINT (0 0)')
<pygeos.Geometry POINT (0 0)>
```

to_geojson(*geometry*, *indent=None*, ***kwargs*)

Converts to the GeoJSON representation of a Geometry.

Note: 'to_geojson' requires at least GEOS 3.10.0.

The GeoJSON format is defined in the RFC 7946. NaN (not-a-number) coordinates will be written as 'null'.

The following are currently unsupported:

- Geometries of type LINEARRING: these are output as 'null'.
- Three-dimensional geometries: the third dimension is ignored.

Parameters

geometry
[str, bytes or array_like]

indent
[int, optional] If indent is a non-negative integer, then GeoJSON will be formatted. An indent level of 0 will only insert newlines. None (the default) selects the most compact representation.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> to_geojson(Geometry("POINT (1 1)"))
'{"type":"Point","coordinates":[1.0,1.0]}'
>>> print(to_geojson(Geometry("POINT (1 1)"), indent=2))
{
  "type": "Point",
  "coordinates": [
      1.0,
      1.0
  ]
}
```

`to_shapely`(*geometry*)

Converts PyGEOS geometries to Shapely.

Warning: When Shapely and PyGEOS are using the same GEOS version, this function assumes that the libraries are actually the same. In some cases (especially when using pip-installed wheels) this may lead to unexpected behaviour. If you require safe (but slower) behaviour, then we recommend setting `pygeos.io.shapely_compatible` to `False`.

Parameters

geometry
[shapely Geometry object or array_like]

Notes

If PyGEOS and Shapely do not use the same GEOS version, the conversion happens through the WKB format and will thus be slower.

Examples

```
>>> to_shapely(Geometry("POINT (1 1)"))
<shapely.geometry.point.Point at 0x7f0c3d737908>
```

`to_wkb`(*geometry*, *hex=False*, *output_dimension=3*, *byte_order=- 1*, *include_srid=False*, ***kwargs*)

Converts to the Well-Known Binary (WKB) representation of a Geometry.

The Well-Known Binary format is defined in the OGC Simple Features Specification for SQL.

The following limitations apply to WKB serialization:

- linearrings will be converted to linestrings

- a point with only NaN coordinates is converted to an empty point
- for GEOS <= 3.7, empty points are always serialized to 3D if output_dimension=3, and to 2D if output_dimension=2
- for GEOS == 3.8, empty points are always serialized to 2D

Parameters

geometry
[Geometry or array_like]

hex
[bool, default False] If true, export the WKB as a hexidecimal string. The default is to return a binary bytes object.

output_dimension
[int, default 3] The output dimension for the WKB. Supported values are 2 and 3. Specifying 3 means that up to 3 dimensions will be written but 2D geometries will still be represented as 2D in the WKB represenation.

byte_order
[int, default -1] Defaults to native machine byte order (-1). Use 0 to force big endian and 1 for little endian.

include_srid
[bool, default False] If True, the SRID is be included in WKB (this is an extension to the OGC WKB specification).

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> to_wkb(Geometry("POINT (1 1)"), byte_order=1)
b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\xf0?'
>>> to_wkb(Geometry("POINT (1 1)"), hex=True, byte_order=1)
'0101000000000000000000F03F000000000000F03F'
```

to_wkt(*geometry*, *rounding_precision=6*, *trim=True*, *output_dimension=3*, *old_3d=False*, ***kwargs*)

Converts to the Well-Known Text (WKT) representation of a Geometry.

The Well-known Text format is defined in the OGC Simple Features Specification for SQL.

The following limitations apply to WKT serialization:

- for GEOS <= 3.8 a multipoint with an empty sub-geometry will raise an exception
- for GEOS <= 3.8 empty geometries are always serialized to 2D
- for GEOS >= 3.9 only simple empty geometries can be 3D, collections are still always 2D

Parameters

geometry
[Geometry or array_like]

rounding_precision
[int, default 6] The rounding precision when writing the WKT string. Set to a value of -1 to indicate the full precision.

trim
[bool, default True] If True, trim unnecessary decimals (trailing zeros).

output_dimension
[int, default 3] The output dimension for the WKT string. Supported values are 2 and 3. Specifying 3 means that up to 3 dimensions will be written but 2D geometries will still be represented as 2D in the WKT string.

old_3d
[bool, default False] Enable old style 3D/4D WKT generation. By default, new style 3D/4D WKT (ie. "POINT Z (10 20 30)") is returned, but with `old_3d=True` the WKT will be formatted in the style "POINT (10 20 30)".

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Notes

The defaults differ from the default of the GEOS library. To mimic this, use:

```
to_wkt(geometry, rounding_precision=-1, trim=False, output_dimension=2)
```

Examples

```
>>> to_wkt(Geometry("POINT (0 0)"))
'POINT (0 0)'
>>> to_wkt(Geometry("POINT (0 0)"), rounding_precision=3, trim=False)
'POINT (0.000 0.000)'
>>> to_wkt(Geometry("POINT (0 0)"), rounding_precision=-1, trim=False)
'POINT (0.0000000000000000 0.0000000000000000)'
>>> to_wkt(Geometry("POINT (1 2 3)"), trim=True)
'POINT Z (1 2 3)'
>>> to_wkt(Geometry("POINT (1 2 3)"), trim=True, output_dimension=2)
'POINT (1 2)'
>>> to_wkt(Geometry("POINT (1 2 3)"), trim=True, old_3d=True)
'POINT (1 2 3)'
```

5.1.5 Measurement

area(*geometry*, ***kwargs*)

Computes the area of a (multi)polygon.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> area(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
100.0
>>> area(Geometry("MULTIPOLYGON (((0 0, 0 10, 10 10, 0 0)), ((0 0, 0 10, 10 10, 0␣
↪0)))"))
100.0
>>> area(Geometry("POLYGON EMPTY"))
0.0
>>> area(None)
nan
```

bounds(*geometry*, ***kwargs*)

Computes the bounds (extent) of a geometry.

For each geometry these 4 numbers are returned: min x, min y, max x, max y.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> bounds(Geometry("POINT (2 3)")).tolist()
[2.0, 3.0, 2.0, 3.0]
>>> bounds(Geometry("LINESTRING (0 0, 0 2, 3 2)")).tolist()
[0.0, 0.0, 3.0, 2.0]
>>> bounds(Geometry("POLYGON EMPTY")).tolist()
[nan, nan, nan, nan]
>>> bounds(None).tolist()
[nan, nan, nan, nan]
```

distance(*a*, *b*, ***kwargs*)

Computes the Cartesian distance between two geometries.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> distance(Geometry("POINT (10 0)"), point)
10.0
>>> distance(Geometry("LINESTRING (1 1, 1 -1)"), point)
1.0
>>> distance(Geometry("POLYGON ((3 0, 5 0, 5 5, 3 5, 3 0))"), point)
3.0
>>> distance(Geometry("POINT EMPTY"), point)
nan
>>> distance(None, point)
nan
```

frechet_distance(*a*, *b*, *densify=None*, ***kwargs*)

Compute the discrete Fréchet distance between two geometries.

Note: 'frechet_distance' requires at least GEOS 3.7.0.

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.

Parameters

a, b
[Geometry or array_like]

densify
[float or array_like, optional] The value of densify is required to be between 0 and 1.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line_1 = Geometry("LINESTRING (0 0, 100 0)")
>>> line_2 = Geometry("LINESTRING (0 0, 50 50, 100 0)")
>>> frechet_distance(line_1, line_2)
70.71...
>>> frechet_distance(line_1, line_2, densify=0.5)
50.0
>>> frechet_distance(line_1, Geometry("LINESTRING EMPTY"))
nan
>>> frechet_distance(line_1, None)
nan
```

hausdorff_distance(*a*, *b*, *densify=None*, ***kwargs*)

Compute the discrete Hausdorff distance between two geometries.

The Hausdorff 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.

Parameters

a, b
[Geometry or array_like]

densify
[float or array_like, optional] The value of densify is required to be between 0 and 1.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line_1 = Geometry("LINESTRING (130 0, 0 0, 0 150)")
>>> line_2 = Geometry("LINESTRING (10 10, 10 150, 130 10)")
>>> hausdorff_distance(line_1, line_2)
14.14...
>>> hausdorff_distance(line_1, line_2, densify=0.5)
70.0
>>> hausdorff_distance(line_1, Geometry("LINESTRING EMPTY"))
nan
>>> hausdorff_distance(line_1, None)
nan
```

length(*geometry*, ***kwargs*)

Computes the length of a (multi)linestring or polygon perimeter.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> length(Geometry("LINESTRING (0 0, 0 2, 3 2)"))
5.0
>>> length(Geometry("MULTILINESTRING ((0 0, 1 0), (0 0, 1 0))"))
2.0
>>> length(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
40.0
>>> length(Geometry("LINESTRING EMPTY"))
0.0
>>> length(None)
nan
```

minimum_bounding_radius(*geometry*, ***kwargs*)

Computes the radius of the minimum bounding circle that encloses an input geometry.

Note: 'minimum_bounding_radius' requires at least GEOS 3.8.0.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

minimum_bounding_circle

Examples

```
>>> minimum_bounding_radius(Geometry("POLYGON ((0 5, 5 10, 10 5, 5 0, 0 5))"))
5.0
>>> minimum_bounding_radius(Geometry("LINESTRING (1 1, 1 10)"))
4.5
>>> minimum_bounding_radius(Geometry("MULTIPOINT (2 2, 4 2)"))
1.0
>>> minimum_bounding_radius(Geometry("POINT (0 1)"))
0.0
>>> minimum_bounding_radius(Geometry("GEOMETRYCOLLECTION EMPTY"))
0.0
```

minimum_clearance(*geometry*, ***kwargs*)

Computes the Minimum Clearance distance.

Note: 'minimum_clearance' requires at least GEOS 3.6.0.

A geometry's "minimum clearance" 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.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> minimum_clearance(Geometry("POLYGON((0 0, 0 10, 5 6, 10 10, 10 0, 5 4, 0 0))"))
2.0
>>> minimum_clearance(Geometry("POLYGON EMPTY"))
inf
>>> minimum_clearance(None)
nan
```

total_bounds(*geometry*, ***kwargs*)

Computes the total bounds (extent) of the geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Returns

numpy ndarray of [xmin, ymin, xmax, ymax]

```
>>> total_bounds(Geometry("POINT (2 3)")).tolist()
    ..
```

[2.0, 3.0, 2.0, 3.0]

```
>>> total_bounds([Geometry("POINT (2 3)"), Geometry("POINT (4 5)")]).
↪tolist()
    ..
```

[2.0, 3.0, 4.0, 5.0]

```
>>> total_bounds([Geometry("LINESTRING (0 1, 0 2, 3 2)"),Geometry(
↪"LINESTRING (4 4, 4 6, 6 7)")]).tolist()
    ..
```

[0.0, 1.0, 6.0, 7.0]

```
>>> total_bounds(Geometry("POLYGON EMPTY")).tolist()
    ..
```

[nan, nan, nan, nan]

```
>>> total_bounds([Geometry("POLYGON EMPTY"), Geometry("POINT (2 3)")]).
↪tolist()
    ..
```

[2.0, 3.0, 2.0, 3.0]

```
>>> total_bounds(None).tolist()
    ..
```

[nan, nan, nan, nan]

5.1.6 Predicates

contains(*a*, *b*, ***kwargs*)

Returns True if geometry B is completely inside geometry A.

A contains B if no points of B lie in the exterior of A and at least one point of the interior of B lies in the interior of A.

Note: following this definition, a geometry does not contain its boundary, but it does contain itself. See `contains_properly` for a version where a geometry does not contain itself.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

within
`contains(A, B) == within(B, A)`

contains_properly
contains with no common boundary points

prepare
improve performance by preparing `a` (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> contains(line, Geometry("POINT (0 0)"))
False
>>> contains(line, Geometry("POINT (0.5 0.5)"))
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> contains(area, Geometry("POINT (0 0)"))
False
>>> contains(area, line)
True
>>> contains(area, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4 2,␣
↪4 4, 2 4, 2 2))")
>>> contains(polygon_with_hole, Geometry("POINT(1 1)"))
True
>>> contains(polygon_with_hole, Geometry("POINT(2 2)"))
```

(continues on next page)

(continued from previous page)

```
False
>>> contains(polygon_with_hole, Geometry("LINESTRING(1 1, 5 5)"))
False
>>> contains(area, area)
True
>>> contains(area, None)
False
```

contains_properly(*a*, *b*, ***kwargs*)

Returns True if geometry B is completely inside geometry A, with no common boundary points.

A contains 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` function, where common points on the boundary are allowed.

Note: this function will prepare the geometries under the hood if needed. You can prepare the geometries in advance to avoid repeated preparation when calling this function multiple times.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

contains
contains which allows common boundary points

prepare
improve performance by preparing `a` (the first argument)

Examples

```
>>> area1 = Geometry("POLYGON((0 0, 3 0, 3 3, 0 3, 0 0))")
>>> area2 = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> area3 = Geometry("POLYGON((1 1, 2 1, 2 2, 1 2, 1 1))")
```

`area1` and `area2` have a common border:

```
>>> contains(area1, area2)
True
>>> contains_properly(area1, area2)
False
```

`area3` is completely inside `area1` with no common border:

```
>>> contains(area1, area3)
True
>>> contains_properly(area1, area3)
True
```

covered_by(*a*, *b*, ***kwargs*)

Returns True if no point in geometry A is outside geometry B.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

covers
covered_by(A, B) == covers(B, A)

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> covered_by(Geometry("POINT (0 0)"), line)
True
>>> covered_by(Geometry("POINT (0.5 0.5)"), line)
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> covered_by(Geometry("POINT (0 0)"), area)
True
>>> covered_by(line, area)
True
>>> covered_by(Geometry("LINESTRING(0 0, 2 2)"), area)
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4 2,␣
↪4 4, 2 4, 2 2))")  # NOQA
>>> covered_by(Geometry("POINT(1 1)"), polygon_with_hole)
True
>>> covered_by(Geometry("POINT(2 2)"), polygon_with_hole)
True
>>> covered_by(Geometry("LINESTRING(1 1, 5 5)"), polygon_with_hole)
False
>>> covered_by(area, area)
True
>>> covered_by(None, area)
False
```

covers(*a*, *b*, ***kwargs*)

Returns True if no point in geometry B is outside geometry A.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

covered_by
covers(A, B) == covered_by(B, A)

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> covers(line, Geometry("POINT (0 0)"))
True
>>> covers(line, Geometry("POINT (0.5 0.5)"))
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> covers(area, Geometry("POINT (0 0)"))
True
>>> covers(area, line)
True
>>> covers(area, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4 2,␣
↪4 4, 2 4, 2 2))")  # NOQA
>>> covers(polygon_with_hole, Geometry("POINT(1 1)"))
True
>>> covers(polygon_with_hole, Geometry("POINT(2 2)"))
True
>>> covers(polygon_with_hole, Geometry("LINESTRING(1 1, 5 5)"))
False
>>> covers(area, area)
True
>>> covers(area, None)
False
```

crosses(*a*, *b*, ***kwargs*)

Returns True if A and B spatially cross.

A crosses B if they have some but not all interior points in common, the intersection is one dimension less than the maximum dimension of A or B, and the intersection is not equal to either A or B.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> # A contains B:
>>> crosses(line, Geometry("POINT (0.5 0.5)"))
False
>>> # A and B intersect at a point but do not share all points:
>>> crosses(line, Geometry("MULTIPOINT ((0 1), (0.5 0.5))"))
True
>>> crosses(line, Geometry("LINESTRING(0 1, 1 0)"))
True
>>> # A is contained by B; their intersection is a line (same dimension):
>>> crosses(line, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> # A contains B:
>>> crosses(area, line)
False
>>> # A and B intersect with a line (lower dimension) but do not share all points:
>>> crosses(area, Geometry("LINESTRING(0 0, 2 2)"))
True
>>> # A contains B:
>>> crosses(area, Geometry("POINT (0.5 0.5)"))
False
>>> # A contains some but not all points of B; they intersect at a point:
>>> crosses(area, Geometry("MULTIPOINT ((2 2), (0.5 0.5))"))
True
```

disjoint(*a*, *b*, ***kwargs*)

Returns True if A and B do not share any point in space.

Disjoint implies that overlaps, touches, within, and intersects are False. Note missing (None) values are never disjoint.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

intersects
`disjoint(A, B) == ~intersects(A, B)`

prepare
improve performance by preparing `a` (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> disjoint(line, Geometry("POINT (0 0)"))
False
>>> disjoint(line, Geometry("POINT (0 1)"))
True
>>> disjoint(line, Geometry("LINESTRING(0 2, 2 0)"))
False
>>> empty = Geometry("GEOMETRYCOLLECTION EMPTY")
>>> disjoint(line, empty)
True
>>> disjoint(empty, empty)
True
>>> disjoint(empty, None)
False
>>> disjoint(None, None)
False
```

dwithin(*a*, *b*, *distance*, ***kwargs*)

Returns True if the geometries are within a given distance.

Note: 'dwithin' requires at least GEOS 3.10.0.

Using this function is more efficient than computing the distance and comparing the result.

Parameters

a, b
[Geometry or array_like]

distance
[float] Negative distances always return False.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

distance
compute the actual distance between A and B

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> point = Geometry("POINT (0.5 0.5)")
>>> dwithin(point, Geometry("POINT (2 0.5)"), 2)
True
>>> dwithin(point, Geometry("POINT (2 0.5)"), [2, 1.5, 1]).tolist()
[True, True, False]
>>> dwithin(point, Geometry("POINT (0.5 0.5)"), 0)
True
>>> dwithin(point, None, 100)
False
```

equals(*a*, *b*, ***kwargs*)

Returns True if A and B are spatially equal.

If A is within B and B is within A, A and B are considered equal. The ordering of points can be different.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

equals_exact
Check if A and B are structurally equal given a specified tolerance.

Examples

```
>>> line = Geometry("LINESTRING(0 0, 5 5, 10 10)")
>>> equals(line, Geometry("LINESTRING(0 0, 10 10)"))
True
>>> equals(Geometry("POLYGON EMPTY"), Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> equals(None, None)
False
```

equals_exact(*a*, *b*, *tolerance=0.0*, ***kwargs*)

Returns True if A and B are structurally equal.

This method uses exact coordinate equality, which requires coordinates to be equal (within specified tolerance) and and in the same order for all components of a geometry. This is in contrast with the `equals` function which uses spatial (topological) equality.

Parameters

a, b
[Geometry or array_like]

tolerance
[float or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

equals
Check if A and B are spatially equal.

Examples

```
>>> point1 = Geometry("POINT(50 50)")
>>> point2 = Geometry("POINT(50.1 50.1)")
>>> equals_exact(point1, point2)
False
>>> equals_exact(point1, point2, tolerance=0.2)
True
>>> equals_exact(point1, None, tolerance=0.2)
False
```

Difference between structucal and spatial equality:

```
>>> polygon1 = Geometry("POLYGON((0 0, 1 1, 0 1, 0 0))")
>>> polygon2 = Geometry("POLYGON((0 0, 0 1, 1 1, 0 0))")
>>> equals_exact(polygon1, polygon2)
False
>>> equals(polygon1, polygon2)
True
```

has_z(*geometry*, ***kwargs*)

Returns True if a geometry has a Z coordinate.

Note that this function returns False if the (first) Z coordinate equals NaN or if the geometry is empty.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

get_coordinate_dimension

Examples

```
>>> has_z(Geometry("POINT (0 0)"))
False
>>> has_z(Geometry("POINT Z (0 0 0)"))
True
>>> has_z(Geometry("POINT Z(0 0 nan)"))
False
```

intersects(*a*, *b*, ***kwargs*)

Returns True if A and B share any portion of space.

Intersects implies that overlaps, touches and within are True.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

disjoint
`intersects(A, B) == ~disjoint(A, B)`

prepare
improve performance by preparing `a` (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> intersects(line, Geometry("POINT (0 0)"))
True
>>> intersects(line, Geometry("POINT (0 1)"))
False
>>> intersects(line, Geometry("LINESTRING(0 2, 2 0)"))
True
>>> intersects(None, None)
False
```

`is_ccw`(*geometry*, ***kwargs*)

Returns True if a linestring or linearring is counterclockwise.

Note: 'is_ccw' requires at least GEOS 3.7.0.

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 function for linestrings is `is_ccw(g) & is_simple(g)` and for linearrings `is_ccw(g) & is_valid(g)`.

Parameters

geometry
[Geometry or array_like] This function will return False for non-linear goemetries and for lines with fewer than 4 points (including the closing point).

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_simple
Checks if a linestring is closed and simple.

is_valid
Checks additionally if the geometry is simple.

Examples

```
>>> is_ccw(Geometry("LINEARRING (0 0, 0 1, 1 1, 0 0)"))
False
>>> is_ccw(Geometry("LINEARRING (0 0, 1 1, 0 1, 0 0)"))
True
>>> is_ccw(Geometry("LINESTRING (0 0, 1 1, 0 1)"))
False
>>> is_ccw(Geometry("POINT (0 0)"))
False
```

is_closed(*geometry*, ***kwargs*)

Returns True if a linestring's first and last points are equal.

Parameters

geometry
[Geometry or array_like] This function will return False for non-linestrings.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_ring
Checks additionally if the geometry is simple.

Examples

```
>>> is_closed(Geometry("LINESTRING (0 0, 1 1)"))
False
>>> is_closed(Geometry("LINESTRING(0 0, 0 1, 1 1, 0 0)"))
True
>>> is_closed(Geometry("POINT (0 0)"))
False
```

is_empty(*geometry*, ***kwargs*)

Returns True if a geometry is an empty point, polygon, etc.

Parameters

geometry
[Geometry or array_like] Any geometry type is accepted.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_missing
checks if the object is a geometry

Examples

```
>>> is_empty(Geometry("POINT EMPTY"))
True
>>> is_empty(Geometry("POINT (0 0)"))
False
>>> is_empty(None)
False
```

is_geometry(*geometry*, ***kwargs*)

Returns True if the object is a geometry

Parameters

geometry
[any object or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_missing
check if an object is missing (None)

is_valid_input
check if an object is a geometry or None

Examples

```
>>> is_geometry(Geometry("POINT (0 0)"))
True
>>> is_geometry(Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> is_geometry(None)
False
>>> is_geometry("text")
False
```

is_missing(*geometry*, ***kwargs*)

Returns True if the object is not a geometry (None)

Parameters

geometry
[any object or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_geometry
check if an object is a geometry

is_valid_input
check if an object is a geometry or None

is_empty
checks if the object is an empty geometry

Examples

```
>>> is_missing(Geometry("POINT (0 0)"))
False
>>> is_missing(Geometry("GEOMETRYCOLLECTION EMPTY"))
False
>>> is_missing(None)
True
>>> is_missing("text")
False
```

is_prepared(*geometry*, ***kwargs*)

Returns True if a Geometry is prepared.

Note that it is not necessary to check if a geometry is already prepared before preparing it. It is more efficient to call `prepare` directly because it will skip geometries that are already prepared.

This function will return False for missing geometries (None).

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_valid_input
check if an object is a geometry or None

prepare
prepare a geometry

Examples

```
>>> geometry = Geometry("POINT (0 0)")
>>> is_prepared(Geometry("POINT (0 0)"))
False
>>> from pygeos import prepare; prepare(geometry);
>>> is_prepared(geometry)
True
>>> is_prepared(None)
False
```

is_ring(*geometry*, ***kwargs*)

Returns True if a linestring is closed and simple.

Parameters

geometry
[Geometry or array_like] This function will return False for non-linestrings.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_closed
Checks only if the geometry is closed.

is_simple
Checks only if the geometry is simple.

Examples

```
>>> is_ring(Geometry("POINT (0 0)"))
False
>>> geom = Geometry("LINESTRING(0 0, 1 1)")
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(False, True, False)
>>> geom = Geometry("LINESTRING(0 0, 0 1, 1 1, 0 0)")
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(True, True, True)
>>> geom = Geometry("LINESTRING(0 0, 1 1, 0 1, 1 0, 0 0)")
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(True, False, False)
```

is_simple(*geometry*, ***kwargs*)

Returns True if a Geometry has no anomalous geometric points, such as self-intersections or self tangency.

Note that polygons and linearrings are assumed to be simple. Use is_valid to check these kind of geometries for self-intersections.

Parameters

geometry
[Geometry or array_like] This function will return False for geometrycollections.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_ring
Checks additionally if the geometry is closed.

is_valid
Checks whether a geometry is well formed.

Examples

```
>>> is_simple(Geometry("POLYGON((1 1, 2 1, 2 2, 1 1))"))
True
>>> is_simple(Geometry("LINESTRING(0 0, 1 1, 0 1, 1 0, 0 0)"))
False
>>> is_simple(None)
False
```

is_valid(*geometry*, ***kwargs*)

Returns True if a geometry is well formed.

Parameters

geometry
[Geometry or array_like] Any geometry type is accepted. Returns False for missing values.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_valid_reason
Returns the reason in case of invalid.

Examples

```
>>> is_valid(Geometry("LINESTRING(0 0, 1 1)"))
True
>>> is_valid(Geometry("POLYGON((0 0, 1 1, 1 2, 1 1, 0 0))"))
False
>>> is_valid(Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> is_valid(None)
False
```

is_valid_input(*geometry*, ***kwargs*)

Returns True if the object is a geometry or None

Parameters

geometry
[any object or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_geometry
checks if an object is a geometry

is_missing
checks if an object is None

Examples

```
>>> is_valid_input(Geometry("POINT (0 0)"))
True
>>> is_valid_input(Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> is_valid_input(None)
True
>>> is_valid_input(1.0)
False
>>> is_valid_input("text")
False
```

is_valid_reason(*geometry*, ***kwargs*)

Returns a string stating if a geometry is valid and if not, why.

Parameters

geometry
[Geometry or array_like] Any geometry type is accepted. Returns None for missing values.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_valid
returns True or False

Examples

```
>>> is_valid_reason(Geometry("LINESTRING(0 0, 1 1)"))
'Valid Geometry'
>>> is_valid_reason(Geometry("POLYGON((0 0, 1 1, 1 2, 1 1, 0 0))"))
'Ring Self-intersection[1 1]'
>>> is_valid_reason(None) is None
True
```

overlaps(*a*, *b*, ***kwargs*)

Returns True if A and B spatially overlap.

A and B overlap if they have some but not all points in common, have the same dimension, and the intersection of the interiors of the two geometries has the same dimension as the geometries themselves. That is, only polyons can overlap other polygons and only lines can overlap other lines.

If either A or B are None, the output is always False.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> poly = Geometry("POLYGON ((0 0, 0 4, 4 4, 4 0, 0 0))")
>>> # A and B share all points (are spatially equal):
>>> overlaps(poly, poly)
False
>>> # A contains B; all points of B are within A:
>>> overlaps(poly, Geometry("POLYGON ((0 0, 0 2, 2 2, 2 0, 0 0))"))
False
>>> # A partially overlaps with B:
>>> overlaps(poly, Geometry("POLYGON ((2 2, 2 6, 6 6, 6 2, 2 2))"))
True
>>> line = Geometry("LINESTRING (2 2, 6 6)")
>>> # A and B are different dimensions; they cannot overlap:
>>> overlaps(poly, line)
False
>>> overlaps(poly, Geometry("POINT (2 2)"))
False
>>> # A and B share some but not all points:
>>> overlaps(line, Geometry("LINESTRING (0 0, 4 4)"))
True
>>> # A and B intersect only at a point (lower dimension); they do not overlap
>>> overlaps(line, Geometry("LINESTRING (6 0, 0 6)"))
False
>>> overlaps(poly, None)
False
>>> overlaps(None, None)
False
```

relate(*a*, *b*, ***kwargs*)

Returns a string representation of the DE-9IM intersection matrix.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> relate(point, line)
'F0FFFF102'
```

relate_pattern(*a*, *b*, *pattern*, ***kwargs*)

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***'`.

Parameters

a, b
[Geometry or array_like]

pattern
[string]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> point = Geometry("POINT (0.5 0.5)")
>>> square = Geometry("POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))")
>>> relate(point, square)
'0FFFFF212'
>>> relate_pattern(point, square, "T*F**F***")
True
```

touches(*a*, *b*, ***kwargs*)

Returns True if the only points shared between A and B are on the boundary of A and B.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

prepare
improve performance by preparing `a` (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 2, 2 0)")
>>> touches(line, Geometry("POINT(0 2)"))
True
>>> touches(line, Geometry("POINT(1 1)"))
False
>>> touches(line, Geometry("LINESTRING(0 0, 1 1)"))
True
>>> touches(line, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> touches(area, Geometry("POINT(0.5 0)"))
True
```

(continues on next page)

(continued from previous page)

```
>>> touches(area, Geometry("POINT(0.5 0.5)"))
False
>>> touches(area, line)
True
>>> touches(area, Geometry("POLYGON((0 1, 1 1, 1 2, 0 2, 0 1))"))
True
```

within(*a*, *b*, ***kwargs*)

Returns True if geometry A is completely inside geometry B.

A is within B if no points of A lie in the exterior of B and at least one point of the interior of A lies in the interior of B.

Parameters

a, b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

contains
within(A, B) == contains(B, A)

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> within(Geometry("POINT (0 0)"), line)
False
>>> within(Geometry("POINT (0.5 0.5)"), line)
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> within(Geometry("POINT (0 0)"), area)
False
>>> within(line, area)
True
>>> within(Geometry("LINESTRING(0 0, 2 2)"), area)
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4 2,␣
↪4 4, 2 4, 2 2))")  # NOQA
>>> within(Geometry("POINT(1 1)"), polygon_with_hole)
True
>>> within(Geometry("POINT(2 2)"), polygon_with_hole)
False
>>> within(Geometry("LINESTRING(1 1, 5 5)"), polygon_with_hole)
False
>>> within(area, area)
True
```

(continues on next page)

(continued from previous page)

```
>>> within(None, area)
False
```

5.1.7 Set operations

coverage_union(*a*, *b*, ***kwargs*)

Merges multiple polygons into one. This is an optimized version of union which assumes the polygons to be non-overlapping.

Note: 'coverage_union' requires at least GEOS 3.8.0.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

coverage_union_all

Examples

```
>>> from pygeos.constructive import normalize
>>> polygon = Geometry("POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))")
>>> normalize(coverage_union(polygon, Geometry("POLYGON ((1 0, 1 1, 2 1, 2 0, 1 0))
↪")))
<pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 2 1, 2 0, 1 0, 0 0))>
```

Union with None returns same polygon >>> normalize(coverage_union(polygon, None)) <pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>

coverage_union_all(*geometries*, *axis=None*, ***kwargs*)

Returns the union of multiple polygons of a geometry collection. This is an optimized version of union which assumes the polygons to be non-overlapping.

Note: 'coverage_union_all' requires at least GEOS 3.8.0.

Parameters

geometries
[array_like]

axis
[int, optional] Axis along which the operation is performed. The default (None) performs the operation over all axes, returning a scalar value. Axis may be negative, in which case it counts from the last to the first axis.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

coverage_union

Examples

```
>>> from pygeos.constructive import normalize
>>> polygon_1 = Geometry("POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))")
>>> polygon_2 = Geometry("POLYGON ((1 0, 1 1, 2 1, 2 0, 1 0))")
>>> normalize(coverage_union_all([polygon_1, polygon_2]))
<pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 2 1, 2 0, 1 0, 0 0))>
```

difference(*a*, *b*, *grid_size=None*, ***kwargs*)

Returns the part of geometry A that does not intersect with geometry B.

If grid_size is nonzero, input coordinates will be snapped to a precision grid of that size and resulting coordinates will be snapped to that same grid. If 0, this operation will use double precision coordinates. If None, the highest precision of the inputs will be used, which may be previously set using set_precision. Note: returned geometry does not have precision set unless specified previously by set_precision.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

grid_size
[float, optional] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

set_precision

Examples

```
>>> from pygeos.constructive import normalize
>>> line = Geometry("LINESTRING (0 0, 2 2)")
>>> difference(line, Geometry("LINESTRING (1 1, 3 3)"))
<pygeos.Geometry LINESTRING (0 0, 1 1)>
>>> difference(line, Geometry("LINESTRING EMPTY"))
<pygeos.Geometry LINESTRING (0 0, 2 2)>
>>> difference(line, None) is None
True
>>> box1 = box(0, 0, 2, 2)
>>> box2 = box(1, 1, 3, 3)
>>> normalize(difference(box1, box2))
<pygeos.Geometry POLYGON ((0 0, 0 2, 1 2, 1 1, 2 1, 2 0, 0 0))>
>>> box1 = box(0.1, 0.2, 2.1, 2.1)
>>> difference(box1, box2, grid_size=1)
<pygeos.Geometry POLYGON ((0 0, 0 2, 1 2, 1 1, 2 1, 2 0, 0 0))>
```

intersection(*a*, *b*, *grid_size=None*, ***kwargs*)

Returns the geometry that is shared between input geometries.

If grid_size is nonzero, input coordinates will be snapped to a precision grid of that size and resulting coordinates will be snapped to that same grid. If 0, this operation will use double precision coordinates. If None, the highest precision of the inputs will be used, which may be previously set using set_precision. Note: returned geometry does not have precision set unless specified previously by set_precision.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

grid_size
[float, optional] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

intersection_all
set_precision

Examples

```
>>> from pygeos.constructive import normalize
>>> line = Geometry("LINESTRING(0 0, 2 2)")
>>> intersection(line, Geometry("LINESTRING(1 1, 3 3)"))
<pygeos.Geometry LINESTRING (1 1, 2 2)>
>>> box1 = box(0, 0, 2, 2)
>>> box2 = box(1, 1, 3, 3)
>>> normalize(intersection(box1, box2))
```

(continues on next page)

(continued from previous page)

```
<pygeos.Geometry POLYGON ((1 1, 1 2, 2 2, 2 1, 1 1))>
>>> box1 = box(0.1, 0.2, 2.1, 2.1)
>>> intersection(box1, box2, grid_size=1)
<pygeos.Geometry POLYGON ((1 1, 1 2, 2 2, 2 1, 1 1))>
```

intersection_all(*geometries*, *axis=None*, ***kwargs*)

Returns the intersection of multiple geometries.

This function ignores None values when other Geometry elements are present. If all elements of the given axis are None, None is returned.

Parameters

geometries
[array_like]

axis
[int, optional] Axis along which the operation is performed. The default (None) performs the operation over all axes, returning a scalar value. Axis may be negative, in which case it counts from the last to the first axis.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc.reduce docs.

See also:

intersection

Examples

```
>>> line_1 = Geometry("LINESTRING(0 0, 2 2)")
>>> line_2 = Geometry("LINESTRING(1 1, 3 3)")
>>> intersection_all([line_1, line_2])
<pygeos.Geometry LINESTRING (1 1, 2 2)>
>>> intersection_all([[line_1, line_2, None]], axis=1).tolist()
[<pygeos.Geometry LINESTRING (1 1, 2 2)>]
```

symmetric_difference(*a*, *b*, *grid_size=None*, ***kwargs*)

Returns the geometry that represents the portions of input geometries that do not intersect.

If grid_size is nonzero, input coordinates will be snapped to a precision grid of that size and resulting coordinates will be snapped to that same grid. If 0, this operation will use double precision coordinates. If None, the highest precision of the inputs will be used, which may be previously set using set_precision. Note: returned geometry does not have precision set unless specified previously by set_precision.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

grid_size
[float, optional] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

symmetric_difference_all
set_precision

Examples

```
>>> from pygeos.constructive import normalize
>>> line = Geometry("LINESTRING(0 0, 2 2)")
>>> symmetric_difference(line, Geometry("LINESTRING(1 1, 3 3)"))
<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>
>>> box1 = box(0, 0, 2, 2)
>>> box2 = box(1, 1, 3, 3)
>>> normalize(symmetric_difference(box1, box2))
<pygeos.Geometry MULTIPOLYGON (((1 2, 1 3, 3 3, 3 1, 2 1, 2 2, 1 2)), ((0 0,...>
>>> box1 = box(0.1, 0.2, 2.1, 2.1)
>>> symmetric_difference(box1, box2, grid_size=1)
<pygeos.Geometry MULTIPOLYGON (((1 2, 1 3, 3 3, 3 1, 2 1, 2 2, 1 2)), ((0 0,...>
```

symmetric_difference_all(*geometries*, *axis=None*, ***kwargs*)

Returns the symmetric difference of multiple geometries.

This function ignores None values when other Geometry elements are present. If all elements of the given axis are None, None is returned.

Parameters

geometries
[array_like]

axis
[int, optional] Axis along which the operation is performed. The default (None) performs the operation over all axes, returning a scalar value. Axis may be negative, in which case it counts from the last to the first axis.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc.reduce docs.

See also:

symmetric_difference

Examples

```
>>> line_1 = Geometry("LINESTRING(0 0, 2 2)")
>>> line_2 = Geometry("LINESTRING(1 1, 3 3)")
>>> symmetric_difference_all([line_1, line_2])
<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>
>>> symmetric_difference_all([[line_1, line_2, None]], axis=1).tolist()
[<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>]
```

union(*a*, *b*, *grid_size=None*, ***kwargs*)

Merges geometries into one.

If grid_size is nonzero, input coordinates will be snapped to a precision grid of that size and resulting coordinates will be snapped to that same grid. If 0, this operation will use double precision coordinates. If None, the highest precision of the inputs will be used, which may be previously set using set_precision. Note: returned geometry does not have precision set unless specified previously by set_precision.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

grid_size
[float, optional] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

union_all
set_precision

Examples

```
>>> from pygeos.constructive import normalize
>>> line = Geometry("LINESTRING(0 0, 2 2)")
>>> union(line, Geometry("LINESTRING(2 2, 3 3)"))
<pygeos.Geometry MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>
>>> union(line, None) is None
True
>>> box1 = box(0, 0, 2, 2)
>>> box2 = box(1, 1, 3, 3)
>>> normalize(union(box1, box2))
<pygeos.Geometry POLYGON ((0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0, 0 0))>
>>> box1 = box(0.1, 0.2, 2.1, 2.1)
>>> union(box1, box2, grid_size=1)
<pygeos.Geometry POLYGON ((0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0, 0 0))>
```

union_all(*geometries*, *grid_size=None*, *axis=None*, ***kwargs*)

Returns the union of multiple geometries.

This function ignores None values when other Geometry elements are present. If all elements of the given axis are None, None is returned.

If grid_size is nonzero, input coordinates will be snapped to a precision grid of that size and resulting coordinates will be snapped to that same grid. If 0, this operation will use double precision coordinates. If None, the highest precision of the inputs will be used, which may be previously set using set_precision. Note: returned geometry does not have precision set unless specified previously by set_precision.

Parameters

geometries
[array_like]

grid_size
[float, optional] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

axis
[int, optional] Axis along which the operation is performed. The default (None) performs the operation over all axes, returning a scalar value. Axis may be negative, in which case it counts from the last to the first axis.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

union
set_precision

Examples

```
>>> from pygeos.constructive import normalize
>>> line_1 = Geometry("LINESTRING(0 0, 2 2)")
>>> line_2 = Geometry("LINESTRING(2 2, 3 3)")
>>> union_all([line_1, line_2])
<pygeos.Geometry MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>
>>> union_all([[line_1, line_2, None]], axis=1).tolist()
[<pygeos.Geometry MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>]
>>> box1 = box(0, 0, 2, 2)
>>> box2 = box(1, 1, 3, 3)
>>> normalize(union_all([box1, box2]))
<pygeos.Geometry POLYGON ((0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0, 0 0))>
>>> box1 = box(0.1, 0.2, 2.1, 2.1)
>>> union_all([box1, box2], grid_size=1)
<pygeos.Geometry POLYGON ((0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0, 0 0))>
```

5.1.8 Constructive operations

boundary(*geometry*, ***kwargs*)

Returns the topological boundary of a geometry.

Parameters

geometry
[Geometry or array_like] This function will return None for geometrycollections.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> boundary(Geometry("POINT (0 0)"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
>>> boundary(Geometry("LINESTRING(0 0, 1 1, 1 2)"))
<pygeos.Geometry MULTIPOINT (0 0, 1 2)>
>>> boundary(Geometry("LINEARRING (0 0, 1 0, 1 1, 0 1, 0 0)"))
<pygeos.Geometry MULTIPOINT EMPTY>
>>> boundary(Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))"))
<pygeos.Geometry LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)>
>>> boundary(Geometry("MULTIPOINT (0 0, 1 2)"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
>>> boundary(Geometry("MULTILINESTRING ((0 0, 1 1), (0 1, 1 0))"))
<pygeos.Geometry MULTIPOINT (0 0, 0 1, 1 0, 1 1)>
>>> boundary(Geometry("GEOMETRYCOLLECTION (POINT (0 0))")) is None
True
```

buffer(*geometry*, *radius*, *quadsegs=8*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Computes the buffer of a geometry for positive and negative buffer radius.

The buffer of a geometry is defined as the Minkowski sum (or difference, for negative width) of the geometry with a circle with radius equal to the absolute value of the buffer radius.

The buffer operation always returns a polygonal result. The negative or zero-distance buffer of lines and points is always empty.

Parameters

geometry
[Geometry or array_like]

width
[float or array_like] Specifies the circle radius in the Minkowski sum (or difference).

quadsegs
[int, default 8] 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 `quadsegs`). Both 'square' and 'flat' result in rectangular line endings, only 'flat' will end at the original vertex, while 'square' involves adding the buffer width.

join_style
[{'round', 'bevel', 'mitre'}, 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 of '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.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> buffer(Geometry("POINT (10 10)"), 2, quadsegs=1)
<pygeos.Geometry POLYGON ((12 10, 10 8, 8 10, 10 12, 12 10))>
>>> buffer(Geometry("POINT (10 10)"), 2, quadsegs=2)
<pygeos.Geometry POLYGON ((12 10, 11.414 8.586, 10 8, 8.586 8.586, 8 10, 8.5...>
>>> buffer(Geometry("POINT (10 10)"), -2, quadsegs=1)
<pygeos.Geometry POLYGON EMPTY>
>>> line = Geometry("LINESTRING (10 10, 20 10)")
>>> buffer(line, 2, cap_style="square")
<pygeos.Geometry POLYGON ((20 12, 22 12, 22 8, 10 8, 8 8, 8 12, 20 12))>
>>> buffer(line, 2, cap_style="flat")
<pygeos.Geometry POLYGON ((20 12, 20 8, 10 8, 10 12, 20 12))>
>>> buffer(line, 2, single_sided=True, cap_style="flat")
<pygeos.Geometry POLYGON ((20 10, 10 10, 10 12, 20 12, 20 10))>
>>> line2 = Geometry("LINESTRING (10 10, 20 10, 20 20)")
>>> buffer(line2, 2, cap_style="flat", join_style="bevel")
<pygeos.Geometry POLYGON ((18 12, 18 20, 22 20, 22 10, 20 8, 10 8, 10 12, 18...>
>>> buffer(line2, 2, cap_style="flat", join_style="mitre")
<pygeos.Geometry POLYGON ((18 12, 18 20, 22 20, 22 8, 10 8, 10 12, 18 12))>
>>> buffer(line2, 2, cap_style="flat", join_style="mitre", mitre_limit=1)
<pygeos.Geometry POLYGON ((18 12, 18 20, 22 20, 21.828 9, 21 8.172, 10 8, 10...>
>>> square = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))")
>>> buffer(square, 2, join_style="mitre")
<pygeos.Geometry POLYGON ((-2 -2, -2 12, 12 12, 12 -2, -2 -2))>
>>> buffer(square, -2, join_style="mitre")
<pygeos.Geometry POLYGON ((2 2, 2 8, 8 8, 8 2, 2 2))>
>>> buffer(square, -5, join_style="mitre")
<pygeos.Geometry POLYGON EMPTY>
>>> buffer(line, float("nan")) is None
True
```

build_area(*geometry*, ***kwargs*)

Creates an areal geometry formed by the constituent linework of given geometry.

Note: 'build_area' requires at least GEOS 3.8.0.

Equivalent of the PostGIS ST_BuildArea() function.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> build_area(Geometry("GEOMETRYCOLLECTION(POLYGON((0 0, 3 0, 3 3, 0 3, 0 0)),␣
↪POLYGON((1 1, 1 2, 2 2, 1 1)))"))
<pygeos.Geometry POLYGON ((0 0, 0 3, 3 3, 3 0, 0 0), (1 1, 2 2, 1 2, 1 1))>
```

centroid(*geometry*, ***kwargs*)

Computes the geometric center (center-of-mass) of a geometry.

For multipoints this is computed as the mean of the input coordinates. For multilinestrings the centroid is weighted by the length of each line segment. For multipolygons the centroid is weighted by the area of each polygon.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> centroid(Geometry("POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))"))
<pygeos.Geometry POINT (5 5)>
>>> centroid(Geometry("LINESTRING (0 0, 2 2, 10 10)"))
<pygeos.Geometry POINT (5 5)>
>>> centroid(Geometry("MULTIPOINT (0 0, 10 10)"))
<pygeos.Geometry POINT (5 5)>
>>> centroid(Geometry("POLYGON EMPTY"))
<pygeos.Geometry POINT EMPTY>
```

clip_by_rect(*geometry*, *xmin*, *ymin*, *xmax*, *ymax*, ***kwargs*)

Returns the portion of a geometry within a rectangle.

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.

Note: empty geometries or geometries that do not overlap with the specified bounds will result in GEOMETRYCOLLECTION EMPTY.

Parameters

geometry
[Geometry or array_like] The geometry to be clipped

xmin
[float] Minimum x value of the rectangle

ymin
[float] Minimum y value of the rectangle

xmax
[float] Maximum x value of the rectangle

ymax
[float] Maximum y value of the rectangle

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line = Geometry("LINESTRING (0 0, 10 10)")
>>> clip_by_rect(line, 0., 0., 1., 1.)
<pygeos.Geometry LINESTRING (0 0, 1 1)>
```

convex_hull(*geometry*, ***kwargs*)

Computes the minimum convex geometry that encloses an input geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> convex_hull(Geometry("MULTIPOINT (0 0, 10 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 10 10, 10 0, 0 0))>
>>> convex_hull(Geometry("POLYGON EMPTY"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

delaunay_triangles(*geometry*, *tolerance=0.0*, *only_edges=False*, ***kwargs*)

Computes a Delaunay triangulation around the vertices of an input geometry.

The output is a geometrycollection containing polygons (default) or linestrings (see only_edges). Returns an None if an input geometry contains less than 3 vertices.

Parameters

geometry
[Geometry or array_like]

tolerance
[float or array_like, default 0.0] Snap input vertices together if their distance is less than this value.

only_edges
[bool or array_like, default False] If set to True, the triangulation will return a collection of linestrings instead of polygons.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> points = Geometry("MULTIPOINT (50 30, 60 30, 100 100)")
>>> delaunay_triangles(points)
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(points, only_edges=True)
<pygeos.Geometry MULTILINESTRING ((50 30, 100 100), (50 30, 60 30), (60 30, ...>
>>> delaunay_triangles(Geometry("MULTIPOINT (50 30, 51 30, 60 30, 100 100)"),␣
↪tolerance=2)
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Geometry("POLYGON ((50 30, 60 30, 100 100, 50 30))"))
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Geometry("LINESTRING (50 30, 60 30, 100 100)"))
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Geometry("GEOMETRYCOLLECTION EMPTY"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

envelope(*geometry*, ***kwargs*)

Computes the minimum bounding box that encloses an input geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> envelope(Geometry("LINESTRING (0 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))>
>>> envelope(Geometry("MULTIPOINT (0 0, 10 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))>
>>> envelope(Geometry("POINT (0 0)"))
<pygeos.Geometry POINT (0 0)>
>>> envelope(Geometry("GEOMETRYCOLLECTION EMPTY"))
<pygeos.Geometry POINT EMPTY>
```

extract_unique_points(*geometry*, ***kwargs*)

Returns all distinct vertices of an input geometry as a multipoint.

Note that only 2 dimensions of the vertices are considered when testing for equality.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> extract_unique_points(Geometry("POINT (0 0)"))
<pygeos.Geometry MULTIPOINT (0 0)>
>>> extract_unique_points(Geometry("LINESTRING(0 0, 1 1, 1 1)"))
<pygeos.Geometry MULTIPOINT (0 0, 1 1)>
>>> extract_unique_points(Geometry("POLYGON((0 0, 1 0, 1 1, 0 0))"))
<pygeos.Geometry MULTIPOINT (0 0, 1 0, 1 1)>
>>> extract_unique_points(Geometry("MULTIPOINT (0 0, 1 1, 0 0)"))
<pygeos.Geometry MULTIPOINT (0 0, 1 1)>
>>> extract_unique_points(Geometry("LINESTRING EMPTY"))
<pygeos.Geometry MULTIPOINT EMPTY>
```

make_valid(*geometry*, ***kwargs*)

Repairs invalid geometries.

Note: 'make_valid' requires at least GEOS 3.8.0.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> make_valid(Geometry("POLYGON((0 0, 1 1, 1 2, 1 1, 0 0))"))
<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (1 1, 1 2))>
```

minimum_bounding_circle(*geometry*, ***kwargs*)

Computes the minimum bounding circle that encloses an input geometry.

Note: 'minimum_bounding_circle' requires at least GEOS 3.8.0.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

minimum_bounding_radius

Examples

```
>>> minimum_bounding_circle(Geometry("POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))"))
<pygeos.Geometry POLYGON ((12.071 5, 11.935 3.621, 11.533 2.294, 10.879 1.07...>
>>> minimum_bounding_circle(Geometry("LINESTRING (1 1, 10 10)"))
<pygeos.Geometry POLYGON ((11.864 5.5, 11.742 4.258, 11.38 3.065, 10.791 1.9...>
>>> minimum_bounding_circle(Geometry("MULTIPOINT (2 2, 4 2)"))
<pygeos.Geometry POLYGON ((4 2, 3.981 1.805, 3.924 1.617, 3.831 1.444, 3.707...>
>>> minimum_bounding_circle(Geometry("POINT (0 1)"))
<pygeos.Geometry POINT (0 1)>
>>> minimum_bounding_circle(Geometry("GEOMETRYCOLLECTION EMPTY"))
<pygeos.Geometry POLYGON EMPTY>
```

normalize(*geometry*, ***kwargs*)

Converts Geometry 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`).

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> p = Geometry("MULTILINESTRING((0 0, 1 1),(2 2, 3 3))")
>>> normalize(p)
<pygeos.Geometry MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

offset_curve(*geometry*, *distance*, *quadsegs=8*, *join_style='round'*, *mitre_limit=5.0*, ***kwargs*)

Returns a (Multi)LineString at a distance from the object on its right or its left side.

For positive distance the offset will be at the left side of the input line and retain the same direction. For a negative distance it will be at the right side and in the opposite direction.

Parameters

geometry
[Geometry or array_like]

distance
[float or array_like] Specifies the offset distance from the input geometry. Negative for right side offset, positive for left side offset.

quadsegs
[int, default 8] Specifies the number of linear segments in a quarter circle in the approximation of circular arcs.

join_style
[{'round', 'bevel', 'mitre'}, 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, default 5.0] Crops of 'mitre'-style joins if the point is displaced from the buffered vertex by more than this limit.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line = Geometry("LINESTRING (0 0, 0 2)")
>>> offset_curve(line, 2)
<pygeos.Geometry LINESTRING (-2 0, -2 2)>
>>> offset_curve(line, -2)
<pygeos.Geometry LINESTRING (2 2, 2 0)>
```

oriented_envelope(*geometry*, ***kwargs*)

Computes the oriented envelope (minimum rotated rectangle) that encloses an input geometry.

Note: 'oriented_envelope' requires at least GEOS 3.6.0.

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.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> oriented_envelope(Geometry("MULTIPOINT (0 0, 10 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 5 -5, 15 5, 10 10, 0 0))>
>>> oriented_envelope(Geometry("LINESTRING (1 1, 5 1, 10 10)"))
<pygeos.Geometry POLYGON ((1 1, 3 -1, 12 8, 10 10, 1 1))>
>>> oriented_envelope(Geometry("POLYGON ((1 1, 15 1, 5 10, 1 1))"))
<pygeos.Geometry POLYGON ((15 1, 15 10, 1 10, 1 1, 15 1))>
>>> oriented_envelope(Geometry("LINESTRING (1 1, 10 1)"))
<pygeos.Geometry LINESTRING (1 1, 10 1)>
>>> oriented_envelope(Geometry("POINT (2 2)"))
<pygeos.Geometry POINT (2 2)>
>>> oriented_envelope(Geometry("GEOMETRYCOLLECTION EMPTY"))
<pygeos.Geometry POLYGON EMPTY>
```

point_on_surface(*geometry*, ***kwargs*)

Returns a point that intersects an input geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> point_on_surface(Geometry("POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))"))
<pygeos.Geometry POINT (5 5)>
>>> point_on_surface(Geometry("LINESTRING (0 0, 2 2, 10 10)"))
<pygeos.Geometry POINT (2 2)>
>>> point_on_surface(Geometry("MULTIPOINT (0 0, 10 10)"))
<pygeos.Geometry POINT (0 0)>
>>> point_on_surface(Geometry("POLYGON EMPTY"))
<pygeos.Geometry POINT EMPTY>
```

polygonize(*geometries*, ***kwargs*)

Creates polygons formed from the linework of a set of Geometries.

Polygonizes an array of Geometries that contain linework which represents the edges of a planar graph. Any type of Geometry 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 result in an empty GeometryCollection. Duplicate segments are ignored.

This function returns the polygons within a GeometryCollection. Individual Polygons can be obtained using `get_geometry` to get a single polygon or `get_parts` to get an array of polygons. MultiPolygons can be constructed from the output using `pygeos.multipolygons(pygeos.get_parts(pygeos.polygonize(geometries)))`.

Parameters

geometries
[array_like] An array of geometries.

axis
[int] Axis along which the geometries are polygonized. The default is to perform a reduction over the last dimension of the input array. A 1D array results in a scalar geometry.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Returns

GeometryCollection or array of GeometryCollections

See also:

get_parts, get_geometry
polygonize_full

Examples

```
>>> lines = [
...     Geometry("LINESTRING (0 0, 1 1)"),
...     Geometry("LINESTRING (0 0, 0 1)"),
...     Geometry("LINESTRING (0 1, 1 1)"),
... ]
>>> polygonize(lines)
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((1 1, 0 0, 0 1, 1 1)))>
```

polygonize_full(*geometries*, ***kwargs*)

Creates polygons formed from the linework of a set of Geometries and return all extra outputs as well.

Polygonizes an array of Geometries that contain linework which represents the edges of a planar graph. Any type of Geometry may be provided as input; only the constituent lines and rings will be used to create the output polygons.

This function performs the same polygonization as `polygonize` but does not only return the polygonal result but all extra outputs as well. The return value consists of 4 elements:

- The polygonal valid output
- **Cut edges**: edges connected on both ends but not part of polygonal output
- **dangles**: edges connected on one end but not part of polygonal output
- **invalid rings**: polygons formed but which are not valid

This function returns the geometries within GeometryCollections. Individual geometries can be obtained using `get_geometry` to get a single geometry or `get_parts` to get an array of geometries.

Parameters

geometries
[array_like] An array of geometries.

axis
[int] Axis along which the geometries are polygonized. The default is to perform a reduction over the last dimension of the input array. A 1D array results in a scalar geometry.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Returns

(polgyons, cuts, dangles, invalid)
tuple of 4 GeometryCollections or arrays of GeometryCollections

See also:

polygonize

Examples

```
>>> lines = [
...     Geometry("LINESTRING (0 0, 1 1)"),
...     Geometry("LINESTRING (0 0, 0 1, 1 1)"),
...     Geometry("LINESTRING (0 1, 1 1)"),
... ]
>>> polygonize_full(lines)
(<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((1 1, 0 0, 0 1, 1 1)))>,
 <pygeos.Geometry GEOMETRYCOLLECTION EMPTY>,
 <pygeos.Geometry GEOMETRYCOLLECTION (LINESTRING (0 1, 1 1))>,
 <pygeos.Geometry GEOMETRYCOLLECTION EMPTY>)
```

reverse(*geometry*, ***kwargs*)

Returns a copy of a Geometry with the order of coordinates reversed.

Note: 'reverse' requires at least GEOS 3.7.0.

If a Geometry is a polygon with interior rings, the interior rings are also reversed.

Points are unchanged. None is returned where Geometry is None.

Parameters

geometry
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

is_ccw
Checks if a Geometry is clockwise.

Examples

```
>>> reverse(Geometry("LINESTRING (0 0, 1 2)"))
<pygeos.Geometry LINESTRING (1 2, 0 0)>
>>> reverse(Geometry("POLYGON ((0 0, 1 0, 1 1, 0 1, 0 0))"))
<pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
>>> reverse(None) is None
True
```

segmentize(*geometry*, *tolerance*, ***kwargs*)

Adds vertices to line segments based on tolerance.

Note: 'segmentize' requires at least GEOS 3.10.0.

Additional vertices will be added to every line segment in an input geometry so that segments are no greater than tolerance. New vertices will evenly subdivide each segment.

Only linear components of input geometries are densified; other geometries are returned unmodified.

Parameters

geometry
[Geometry or array_like]

tolerance
[float or array_like] Additional vertices will be added so that all line segments are no greater than this value. Must be greater than 0.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line = Geometry("LINESTRING (0 0, 0 10)")
>>> segmentize(line, tolerance=5)
<pygeos.Geometry LINESTRING (0 0, 0 5, 0 10)>
>>> poly = Geometry("POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))")
>>> segmentize(poly, tolerance=5)
<pygeos.Geometry POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
>>> segmentize(None, tolerance=5) is None
True
```

simplify(*geometry*, *tolerance*, *preserve_topology=False*, ***kwargs*)

Returns a simplified version of an input geometry using the Douglas-Peucker algorithm.

Parameters

geometry
[Geometry or array_like]

tolerance
[float or array_like] The maximum allowed geometry displacement. The higher this value, the smaller the number of vertices in the resulting geometry.

preserve_topology
[bool, default False] If set to True, the operation will avoid creating invalid geometries.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line = Geometry("LINESTRING (0 0, 1 10, 0 20)")
>>> simplify(line, tolerance=0.9)
<pygeos.Geometry LINESTRING (0 0, 1 10, 0 20)>
>>> simplify(line, tolerance=1)
<pygeos.Geometry LINESTRING (0 0, 0 20)>
>>> polygon_with_hole = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2 4,␣
↪4 4, 4 2, 2 2))")
>>> simplify(polygon_with_hole, tolerance=4, preserve_topology=True)
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2 4, 4 4, 4 2...>
>>> simplify(polygon_with_hole, tolerance=4, preserve_topology=False)
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
```

snap(*geometry*, *reference*, *tolerance*, ***kwargs*)

Snaps an input geometry to reference geometry's vertices.

The tolerance is used to control where snapping is performed. The result geometry is the input geometry with the vertices snapped. If no snapping occurs then the input geometry is returned unchanged.

Parameters

geometry
[Geometry or array_like]

reference
[Geometry or array_like]

tolerance
[float or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> point = Geometry("POINT (0 2)")
>>> snap(Geometry("POINT (0.5 2.5)"), point, tolerance=1)
<pygeos.Geometry POINT (0 2)>
>>> snap(Geometry("POINT (0.5 2.5)"), point, tolerance=0.49)
<pygeos.Geometry POINT (0.5 2.5)>
>>> polygon = Geometry("POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))")
>>> snap(polygon, Geometry("POINT (8 10)"), tolerance=5)
<pygeos.Geometry POLYGON ((0 0, 0 10, 8 10, 10 0, 0 0))>
>>> snap(polygon, Geometry("LINESTRING (8 10, 8 0)"), tolerance=5)
<pygeos.Geometry POLYGON ((0 0, 0 10, 8 10, 8 0, 0 0))>
```

voronoi_polygons(*geometry*, *tolerance=0.0*, *extend_to=None*, *only_edges=False*, ***kwargs*)

Computes a Voronoi diagram from the vertices of an input geometry.

The output is a geometrycollection containing polygons (default) or linestrings (see only_edges). Returns empty if an input geometry contains less than 2 vertices or if the provided extent has zero area.

Parameters

geometry
[Geometry or array_like]

tolerance
[float or array_like, default 0.0] Snap input vertices together if their distance is less than this value.

extend_to
[Geometry or array_like, optional] If provided, the diagram will be extended to cover the envelope of this geometry (unless this envelope is smaller than the input geometry).

only_edges
[bool or array_like, default False] If set to True, the triangulation will return a collection of linestrings instead of polygons.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> from pygeos import normalize
>>> points = Geometry("MULTIPOINT (2 2, 4 2)")
>>> normalize(voronoi_polygons(points))
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((3 0, 3 4, 6 4, 6 0, 3 0)), PO...>
>>> voronoi_polygons(points, only_edges=True)
<pygeos.Geometry LINESTRING (3 4, 3 0)>
>>> voronoi_polygons(Geometry("MULTIPOINT (2 2, 4 2, 4.2 2)"), 0.5, only_edges=True)
<pygeos.Geometry LINESTRING (3 4.2, 3 -0.2)>
>>> voronoi_polygons(points, extend_to=Geometry("LINESTRING (0 0, 10 10)"), only_
↪edges=True)
<pygeos.Geometry LINESTRING (3 10, 3 0)>
>>> voronoi_polygons(Geometry("LINESTRING (2 2, 4 2)"), only_edges=True)
<pygeos.Geometry LINESTRING (3 4, 3 0)>
>>> voronoi_polygons(Geometry("POINT (2 2)"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

5.1.9 Linestring operations

line_interpolate_point(*line*, *distance*, *normalized=False*, ***kwargs*)

Returns a point interpolated at given distance on a line.

Parameters

line
: [Geometry or array_like] For multilinestrings or geometrycollections, the first geometry is taken and the rest is ignored. This function raises a TypeError for non-linear geometries. For empty linear geometries, empty points are returned.

distance
: [float or array_like] Negative values measure distance from the end of the line. Out-of-range values will be clipped to the line endings.

normalized
: [bool, default False] If True, the distance is a fraction of the total line length instead of the absolute distance.

****kwargs**
: For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line = Geometry("LINESTRING(0 2, 0 10)")
>>> line_interpolate_point(line, 2)
<pygeos.Geometry POINT (0 4)>
>>> line_interpolate_point(line, 100)
<pygeos.Geometry POINT (0 10)>
>>> line_interpolate_point(line, -2)
<pygeos.Geometry POINT (0 8)>
>>> line_interpolate_point(line, [0.25, -0.25], normalized=True).tolist()
[<pygeos.Geometry POINT (0 4)>, <pygeos.Geometry POINT (0 8)>]
```

(continues on next page)

(continued from previous page)

```
>>> line_interpolate_point(Geometry("LINESTRING EMPTY"), 1)
<pygeos.Geometry POINT EMPTY>
```

line_locate_point(*line*, *other*, *normalized=False*, ***kwargs*)

Returns the distance to the line origin of given point.

If given point does not intersect with the line, the point will first be projected onto the line after which the distance is taken.

Parameters

line
[Geometry or array_like]

point
[Geometry or array_like]

normalized
[bool, default False] If True, the distance is a fraction of the total line length instead of the absolute distance.

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line = Geometry("LINESTRING(0 2, 0 10)")
>>> line_locate_point(line, Geometry("POINT(4 4)"))
2.0
>>> line_locate_point(line, Geometry("POINT(4 4)"), normalized=True)
0.25
>>> line_locate_point(line, Geometry("POINT(0 18)"))
8.0
>>> line_locate_point(Geometry("LINESTRING EMPTY"), Geometry("POINT(4 4)"))
nan
```

line_merge(*line*, ***kwargs*)

Returns (multi)linestrings formed by combining the lines in a multilinestrings.

Parameters

line
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> line_merge(Geometry("MULTILINESTRING((0 2, 0 10), (0 10, 5 10))"))
<pygeos.Geometry LINESTRING (0 2, 0 10, 5 10)>
>>> line_merge(Geometry("MULTILINESTRING((0 2, 0 10), (0 11, 5 10))"))
<pygeos.Geometry MULTILINESTRING ((0 2, 0 10), (0 11, 5 10))>
>>> line_merge(Geometry("LINESTRING EMPTY"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

shared_paths(*a*, *b*, ***kwargs*)

Returns the shared paths between geom1 and geom2.

Both geometries should be linestrings or arrays of linestrings. A geometrycollection or array 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.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

Examples

```
>>> geom1 = Geometry("LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)")
>>> geom2 = Geometry("LINESTRING (1 0, 2 0, 2 1, 1 1, 1 0)")
>>> shared_paths(geom1, geom2)
<pygeos.Geometry GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MULTILINESTRING ...>
```

shortest_line(*a*, *b*, ***kwargs*)

Returns the shortest 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.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

****kwargs**
For other keyword-only arguments, see the NumPy ufunc docs.

See also:

prepare
improve performance by preparing a (the first argument) (for GEOS>=3.9)

Examples

```
>>> geom1 = Geometry("LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)")
>>> geom2 = Geometry("LINESTRING (0 3, 3 0, 5 3)")
>>> shortest_line(geom1, geom2)
<pygeos.Geometry LINESTRING (1 1, 1.5 1.5)>
```

5.1.10 Coordinate operations

apply(*geometry*, *transformation*, *include_z=False*)

Returns a copy of a geometry array with a function applied to its coordinates.

With the default of `include_z=False`, all returned geometries will be two-dimensional; the third dimension will be discarded, if present. When specifying `include_z=True`, the returned geometries preserve the dimensionality of the respective input geometries.

Parameters

geometry
[Geometry or array_like]

transformation
[function] 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.

Examples

```
>>> apply(Geometry("POINT (0 0)"), lambda x: x + 1)
<pygeos.Geometry POINT (1 1)>
>>> apply(Geometry("LINESTRING (2 2, 4 4)"), lambda x: x * [2, 3])
<pygeos.Geometry LINESTRING (4 6, 8 12)>
>>> apply(None, lambda x: x) is None
True
>>> apply([Geometry("POINT (0 0)"), None], lambda x: x).tolist()
[<pygeos.Geometry POINT (0 0)>, None]
```

By default, the third dimension is ignored:

```
>>> apply(Geometry("POINT Z (0 0 0)"), lambda x: x + 1)
<pygeos.Geometry POINT (1 1)>
>>> apply(Geometry("POINT Z (0 0 0)"), lambda x: x + 1, include_z=True)
<pygeos.Geometry POINT Z (1 1 1)>
```

count_coordinates(*geometry*)

Counts the number of coordinate pairs in a geometry array.

Parameters

geometry
[Geometry or array_like]

Examples

```
>>> count_coordinates(Geometry("POINT (0 0)"))
1
>>> count_coordinates(Geometry("LINESTRING (2 2, 4 4)"))
2
>>> count_coordinates(None)
0
>>> count_coordinates([Geometry("POINT (0 0)"), None])
1
```

get_coordinates(*geometry*, *include_z=False*, *return_index=False*)

Gets coordinates from a geometry array as an array of floats.

The shape of the returned array 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 array is (N, 3).

Parameters

geometry
[Geometry or array_like]

include_z
[bool, default False] If, True include the third dimension in the output. If a geometry has no third dimension, the z-coordinates will be NaN.

return_index
[bool, default False] If True, also return the index of each returned geometry as a separate ndarray of integers. For multidimensional arrays, this indexes into the flattened array (in C contiguous order).

Examples

```
>>> get_coordinates(Geometry("POINT (0 0)")).tolist()
[[0.0, 0.0]]
>>> get_coordinates(Geometry("LINESTRING (2 2, 4 4)")).tolist()
[[2.0, 2.0], [4.0, 4.0]]
>>> get_coordinates(None)
array([], shape=(0, 2), dtype=float64)
```

By default the third dimension is ignored:

```
>>> get_coordinates(Geometry("POINT Z (0 0 0)")).tolist()
[[0.0, 0.0]]
>>> get_coordinates(Geometry("POINT Z (0 0 0)"), include_z=True).tolist()
[[0.0, 0.0, 0.0]]
```

When return_index=True, indexes are returned also:

```
>>> geometries = [Geometry("LINESTRING (2 2, 4 4)"), Geometry("POINT (0 0)")]
>>> coordinates, index = get_coordinates(geometries, return_index=True)
>>> coordinates.tolist(), index.tolist()
([[2.0, 2.0], [4.0, 4.0], [0.0, 0.0]], [0, 0, 1])
```

set_coordinates(*geometry*, *coordinates*)

Adapts the coordinates of a geometry array in-place.

If the coordinates array has shape (N, 2), all returned geometries will be two-dimensional, and the third dimension will be discarded, if present. If the coordinates array has shape (N, 3), the returned geometries preserve the dimensionality of the input geometries.

Warning: The geometry array is modified in-place! If you do not want to modify the original array, you can do `set_coordinates(arr.copy(), newcoords)`.

Parameters

geometry
[Geometry or array_like]

coordinates: array_like

See also:

apply
Returns a copy of a geometry array with a function applied to its coordinates.

Examples

```
>>> set_coordinates(Geometry("POINT (0 0)"), [[1, 1]])
<pygeos.Geometry POINT (1 1)>
>>> set_coordinates([Geometry("POINT (0 0)"), Geometry("LINESTRING (0 0, 0 0)")],␣
↪[[1, 2], [3, 4], [5, 6]]).tolist()
[<pygeos.Geometry POINT (1 2)>, <pygeos.Geometry LINESTRING (3 4, 5 6)>]
>>> set_coordinates([None, Geometry("POINT (0 0)")], [[1, 2]]).tolist()
[None, <pygeos.Geometry POINT (1 2)>]
```

Third dimension of input geometry is discarded if coordinates array does not include one:

```
>>> set_coordinates(Geometry("POINT Z (0 0 0)"), [[1, 1]])
<pygeos.Geometry POINT (1 1)>
>>> set_coordinates(Geometry("POINT Z (0 0 0)"), [[1, 1, 1]])
<pygeos.Geometry POINT Z (1 1 1)>
```

5.1.11 STRTree

class STRtree(*geometries*, *leafsize=10*)

A query-only R-tree created using the Sort-Tile-Recursive (STR) algorithm.

For two-dimensional spatial data. The tree is constructed directly at initialization.

Parameters

geometries
[array_like]

leafsize
[int, default 10] the maximum number of child nodes that a node can have

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> # Query geometries that overlap envelope of input geometries:
>>> tree.query(pygeos.box(2, 2, 4, 4)).tolist()
[2, 3, 4]
>>> # Query geometries that are contained by input geometry:
>>> tree.query(pygeos.box(2, 2, 4, 4), predicate='contains').tolist()
[3]
>>> # Query geometries that overlap envelopes of ``geoms``
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)]).tolist()
[[0, 0, 0, 1, 1], [2, 3, 4, 5, 6]]
>>> tree.nearest([pygeos.points(1,1), pygeos.points(3,5)]).tolist()
[[0, 1], [1, 4]]
```

nearest(*geometry*)

Returns the index of the nearest item in the tree for each input geometry.

Note: 'nearest' requires at least GEOS 3.6.0.

If there are multiple equidistant or intersected geometries in the tree, only a single result is returned for each input geometry, based on the order that tree geometries are visited; this order may be nondeterministic.

Any geometry that is None or empty in the input geometries is omitted from the output.

Parameters

geometry
[Geometry or array_like] Input geometries to query the tree.

Returns

ndarray with shape (2, n)
The first subarray contains input geometry indexes. The second subarray contains tree geometry indexes.

See also:

nearest_all
returns all equidistant geometries and optional distances

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> tree.nearest(pygeos.points(1,1)).tolist()
[[0], [1]]
>>> tree.nearest([pygeos.box(1,1,3,3)]).tolist()
[[0], [1]]
>>> points = pygeos.points(0.5,0.5)
>>> tree.nearest([None, pygeos.points(10,10)]).tolist()
[[1], [9]]
```

nearest_all(*geometry*, *max_distance=None*, *return_distance=False*)

Returns the index of the nearest item(s) in the tree for each input geometry.

Note: 'nearest_all' requires at least GEOS 3.6.0.

If there are multiple equidistant or intersected geometries in tree, all are returned. Tree indexes are returned in the order they are visited for each input geometry and may not be in ascending index order; no meaningful order is implied.

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. Only those input geometries with at least one tree item within +/- max_distance beyond their envelope will be evaluated.

The distance, if returned, will be 0 for any intersected geometries in the tree.

Any geometry that is None or empty in the input geometries is omitted from the output.

Parameters

geometry
[Geometry or array_like] Input geometries to query the tree.

max_distance
[float, optional] Maximum distance within which to query for nearest items in tree. Must be greater than 0.

return_distance
[bool, default False] If True, will return distances in addition to indexes.

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.

See also:

nearest
returns singular nearest geometry for each input

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> tree.nearest_all(pygeos.points(1,1)).tolist()
[[0], [1]]
>>> tree.nearest_all([pygeos.box(1,1,3,3)]).tolist()
[[0, 0, 0], [1, 2, 3]]
>>> points = pygeos.points(0.5,0.5)
>>> index, distance = tree.nearest_all(points, return_distance=True)
>>> index.tolist()
[[0, 0], [0, 1]]
>>> distance.round(4).tolist()
[0.7071, 0.7071]
```

(continues on next page)

(continued from previous page)

```
>>> tree.nearest_all(None).tolist()
[[], []]
```

query(*geometry*, *predicate=None*, *distance=None*)

Return the index of all geometries in the tree with extents that intersect the envelope of the input geometry.

If predicate is provided, a prepared version of the input geometry is tested using the predicate function against each item whose extent intersects the envelope of the input geometry: predicate(geometry, tree_geometry).

The 'dwithin' predicate requires GEOS >= 3.10.

If geometry is None, an empty array is returned.

Parameters

geometry
[Geometry] The envelope of the geometry is taken automatically for querying the tree.

predicate
[{None, 'intersects', 'within', 'contains', 'overlaps', 'crosses','touches', 'covers', 'covered_by', 'contains_properly', 'dwithin'}, optional] The predicate to use for testing geometries from the tree that are within the input geometry's envelope.

distance
[number, optional] Distance around the geometry within which to query the tree for the 'dwithin' predicate. Required if predicate='dwithin'.

Returns

ndarray
Indexes of geometries in tree

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> tree.query(pygeos.box(1,1, 3,3)).tolist()
[1, 2, 3]
>>> # Query geometries that are contained by input geometry
>>> tree.query(pygeos.box(2, 2, 4, 4), predicate='contains').tolist()
[3]
>>> # Query geometries within 1 unit distance of input geometry
>>> tree.query(pygeos.points(0.5, 0.5), predicate='dwithin', distance=1.0).
↪tolist()
[0, 1]
```

query_bulk(*geometry*, *predicate=None*, *distance=None*)

Returns all combinations of each input geometry and geometries in the tree where the envelope of each input geometry intersects with the envelope of a tree geometry.

If predicate is provided, a prepared version of each input geometry is tested using the predicate function against each item whose extent intersects the envelope of the input geometry: predicate(geometry, tree_geometry).

The 'dwithin' predicate requires GEOS >= 3.10.

This returns an array with shape (2,n) where the subarrays correspond to the indexes of the input geometries and indexes of the tree geometries associated with each. To generate an array of pairs of input geometry index and tree geometry index, simply transpose the results.

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 envelope overlap or optional predicate are returned.

Any geometry that is None or empty in the input geometries is omitted from the output.

Parameters

geometry
[Geometry or array_like] Input geometries to query the tree. The envelope of each geometry is automatically calculated for querying the tree.

predicate
[{None, 'intersects', 'within', 'contains', 'overlaps', 'crosses','touches', 'covers', 'covered_by', 'contains_properly', 'dwithin'}, optional] The predicate to use for testing geometries from the tree that are within the input geometry's envelope.

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 (2, n)
The first subarray contains input geometry indexes. The second subarray contains tree geometry indexes.

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)]).tolist()
[[0, 0, 0, 1, 1], [2, 3, 4, 5, 6]]
>>> # Query for geometries that contain tree geometries
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)], predicate=
↪'contains').tolist()
[[0], [3]]
>>> # To get an array of pairs of index of input geometry, index of tree␣
↪geometry,
>>> # transpose the output:
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)]).T.tolist()
[[0, 2], [0, 3], [0, 4], [1, 5], [1, 6]]
>>> # Query for tree geometries within 1 unit distance of input geometries
>>> tree.query_bulk([pygeos.points(0.5, 0.5)], predicate='dwithin', distance=1.
↪0).tolist()
[[0, 0], [0, 1]]
```

5.1.12 Testing

The functions in this module are not directly importable from the root `pygeos` module. Instead, import them from the submodule as follows:

```
>>> from pygeos.testing import assert_geometries_equal
```

assert_geometries_equal(*x*, *y*, *tolerance=1e-07*, *equal_none=True*, *equal_nan=True*, *normalize=False*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two geometry array_like objects are not equal.

Given two array_like objects, check that the shape is equal and all elements of these objects are equal. An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in pygeos, no assertion is raised if both objects have NaNs/Nones in the same positions.

Parameters

x
[Geometry or array_like]

y
[Geometry or array_like]

equal_none
[bool, default True] Whether to consider None elements equal to other None elements.

equal_nan
[bool, default True] Whether to consider nan coordinates as equal to other nan coordinates.

normalize
[bool, default False] Whether to normalize geometries prior to comparison.

err_msg
[str, optional] The error message to be printed in case of failure.

verbose
[bool, optional] If True, the conflicting values are appended to the error message.

5.1.13 Changelog

Version 0.14 (2022-12-12)

- Added support for Python 3.11 binary wheels (#458).
- Dropped support for Python 3.6 (#458).
- Binary wheels now include GEOS 3.10.4 (#458).
- Fixed unittests for GEOS 3.11 (#458).
- Only use the 'fast' path in to/from_shapely when using conda (#459).

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Casper van der Wel
- Joris Van den Bossche

Version 0.13 (2022-08-25)

Distribution

- All binary wheels now have GEOS 3.10.3. See https://github.com/libgeos/geos/blob/3.10/NEWS for the changes (#454).

Bug fixes

- Fixed the `to_shapely` and `from_shapely` functions for compatibility with the upcoming Shapely 2.0 release (#452)

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Casper van der Wel
- Joris Van den Bossche
- Mike Taves

Version 0.12 (2021-12-03)

Distribution

- Distribute binary wheels for Apple Silicon architecture (arm64 and universal2) (#427).
- Removed 32-bit architecture wheels for Python 3.10 (#427).
- All binary wheels now have GEOS 3.10.1. See https://github.com/libgeos/geos/blob/main/NEWS for the changes (#422).
- Linux x86_64 and i686 wheels are now built using the manylinux2014 image instead of manylinux2010 (#445).

Major enhancements

- Added `pygeos.dwithin` for GEOS >= 3.10 (#417).
- Added `dwithin` predicate to `STRtree` `query` and `query_bulk` methods to find geometries within a search distance for GEOS >= 3.10 (#425).
- Added GeoJSON input/output capabilities (`pygeos.from_geojson`, `pygeos.to_geojson`) for GEOS >= 3.10 (#413).
- Performance improvement in constructing LineStrings or LinearRings from numpy arrays for GEOS >= 3.10 (#436)

API Changes

- When constructing a linearring through `pygeos.linearrings` or a polygon through `pygeos.polygons` the ring is automatically closed when supplied with 3 coordinates also when the first and last are already equal (#431).

Bug fixes

- Raise `GEOSException` in the rare case when predicate evalution in `STRtree.query` errors. Previously, the exceptions were ignored silently and the geometry was added to the result (as if the predicate returned `True`) (#432).
- Hide `RuntimeWarning` when using `total_bounds` on empty geometries, empty arrays, or geometries with NaN coordinates (#441).

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche

Version 0.11.1 (2021-10-30)

Distribution

- Distribute binary wheels for Python 3.10 (#416).
- All binary wheels now have GEOS 3.10.0. See https://github.com/libgeos/geos/blob/main/NEWS for the changes (#416).

Major enhancements

- Optionally output to a user-specified array (`out` keyword argument) when constructing geometries from `indices` (#380).
- Added `pygeos.empty` to create a geometry array pre-filled with None or with empty geometries (#381).
- Added `pygeos.force_2d` and `pygeos.force_3d` to change the dimensionality of the coordinates in a geometry (#396).
- Added `pygeos.testing.assert_geometries_equal` (#401).

API Changes

- The default behaviour of `pygeos.set_precision` is now to always return valid geometries. Before, the default was `preserve_topology=False` which caused confusion because it mapped to GEOS_PREC_NO_TOPO (the new 'pointwise'). At the same time, GEOS < 3.10 implementation was not entirely correct so that some geometries did and some did not preserve topology with this mode. Now, the new `mode` argument controls the behaviour and the `preserve_topology` argument is deprecated (#410).
- When constructing a linearring through `pygeos.linearrings` or a polygon through `pygeos.polygons` now a `ValueError` is raised (instead of a `GEOSException`) if the ring contains less than 4 coordinates including ring closure (#378).
- Removed deprecated `normalize` keyword argument in `pygeos.line_locate_point` and `pygeos.line_interpolate_point` (#410).

Bug fixes

- Return True instead of False for LINEARRING geometries in `is_closed` (#379).
- Fixed the WKB serialization of 3D empty points for GEOS >= 3.9.0 (#392).
- Fixed the WKT serialization of single part 3D empty geometries for GEOS >= 3.9.0 (#402).
- Fixed the WKT serialization of multipoints with empty points for GEOS >= 3.9.0 (#392).
- Fixed a segfault when getting coordinates from empty points in GEOS 3.8.0 (#415).

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward

- Casper van der Wel
- Joris Van den Bossche

Version 0.10.2 (2021-08-23)

Distribution

Unittests are now included in the pygeos distribution. Run them by 1) installing `pytest` (or `pygeos[test]`) and 2) invoking `pytest --pyargs pygeos.tests`.

We started using a new tool for building binary wheels: `cibuildwheel`. This resulted into the following improvements in the distributed binary wheels:

- Windows binary wheels now contain mangled DLLs, which avoids conflicts with other GEOS versions present on the system (a.k.a. 'DLL hell') (#365).
- Windows binary wheels now contain the Microsoft Visual C++ Runtime Files (msvcp140.dll) for usage on systems without the C++ redistributable (#365).
- Linux x86_64 and i686 wheels are now built using the manylinux2010 image instead of manylinux1 (#365).
- Linux aarch64 wheels are now available for Python 3.9 (manylinux2014, #365).

Bug fixes

- Fixed operations on geometry arrays containing NULL instead of None. These occur for instance by using `numpy.empty_like` (#371)

Acknowledgements

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche

Version 0.10.1 (2021-07-06)

Bug fixes

- Fixed the `box` and `set_precision` functions with numpy 1.21 (#367).
- Fixed `STRtree` creation to allow querying the tree in a multi-threaded context (#361).

Acknowledgements

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche

Version 0.10 (2021-05-18)

Major enhancements

- Addition of `nearest` and `nearest_all` functions to `STRtree` for GEOS >= 3.6 to find the nearest neighbors (#272).
- Enable bulk construction of geometries with different number of coordinates by optionally taking index arrays in all creation functions (#230, #322, #326, #346).
- Released the GIL in all geometry creation functions (#310, #326).
- Added the option to return the geometry index in `get_coordinates` (#318).
- Added the `get_rings` function, similar as `get_parts` but specifically to extract the rings of Polygon geometries (#342).
- Updated `box` ufunc to use internal C function for creating polygon (about 2x faster) and added `ccw` parameter to create polygon in counterclockwise (default) or clockwise direction (#308).
- Added `to_shapely` and improved performance of `from_shapely` in the case GEOS versions are different (#312).

API Changes

- STRtree default leaf size is now 10 instead of 5, for somewhat better performance under normal conditions (#286)
- Deprecated `VALID_PREDICATES` set from `pygeos.strtree` package; these can be constructed in downstream libraries using the `pygeos.strtree.BinaryPredicate` enum. This will be removed in a future release.
- `points`, `linestrings`, `linearrings`, and `polygons` now return a `GEOSException` instead of a `ValueError` or `TypeError` for invalid input (#310, #326).
- Addition of `on_invalid` parameter to `from_wkb` and `from_wkt` to optionally return invalid WKB geometries as `None`.
- Removed the (internal) function `lib.polygons_without_holes` and renamed `lib.polygons_with_holes` to `lib.polygons` (#326).
- `polygons` will now return an empty polygon for *None* inputs (#346).
- Removed compatibility with Python 3.5 (#341).

Added GEOS functions

- Addition of a `contains_properly` function (#267)
- Addition of a `polygonize` function (#275)
- Addition of a `polygonize_full` function (#298)
- Addition of a `segmentize` function for GEOS >= 3.10 (#299)
- Addition of `oriented_envelope` and `minimum_rotated_rectangle` functions (#314)
- Addition of `minimum_bounding_circle` and `minimum_bounding_radius` functions for GEOS >= 3.8 (#315)
- Addition of a `shortest_line` ("nearest points") function (#334)

Bug fixes

- Fixed portability issue for ARM architecture (#293)
- Fixed segfault in `linearrings` and `box` when constructing a geometry with nan coordinates (#310).
- Fixed segfault in `polygons` (with holes) when None was provided.
- Fixed memory leak in `polygons` when non-linearring input was provided.

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche
- Martin Fleischmann
- Mike Taves
- Tanguy Ophoff +
- James Myatt +

Version 0.9 (2021-01-23)

Major enhancements

- Addition of `prepare` function that generates a GEOS prepared geometry which is stored on the Geometry object itself. All binary predicates (except `equals`) make use of this. Helper functions `destroy_prepared` and `is_prepared` are also available. (#92, #252)
- Use previously prepared geometries within `STRtree` `query` and `query_bulk` functions if available (#246)
- Official support for Python 3.9 and numpy 1.20 (#278, #279)
- Drop support for Python 3.5 (#211)
- Added support for pickling to `Geometry` objects (#190)
- The `apply` function for coordinate transformations and the `set_coordinates` function now support geometries with z-coordinates (#131)
- Addition of Cython and internal PyGEOS C API to enable easier development of internal functions (previously all significant internal functions were developed in C). Added a Cython-implemented `get_parts` function (#51)

API Changes

- Geometry and counting functions (`get_num_coordinates`, `get_num_geometries`, `get_num_interior_rings`, `get_num_points`) now return 0 for `None` input values instead of -1 (#218)
- `intersection_all` and `symmetric_difference_all` now ignore None values instead of returning None if any value is None (#249)
- `union_all` now returns None (instead of `GEOMETRYCOLLECTION EMPTY`) if all input values are None (#249)
- The default axis of `union_all`, `intersection_all`, `symmetric_difference_all`, and `coverage_union_all` can now reduce over multiple axes. The default changed from the first axis (`0`) to all axes (`None`) (#266)
- Argument in `line_interpolate_point` and `line_locate_point` was renamed from `normalize` to `normalized` (#209)
- Addition of `grid_size` parameter to specify fixed-precision grid for `difference`, `intersection`, `symmetric_difference`, `union`, and `union_all` operations for GEOS >= 3.9 (#276)

Added GEOS functions

- Release the GIL for `is_geometry()`, `is_missing()`, and `is_valid_input()` (#207)
- Addition of a `is_ccw()` function for GEOS >= 3.7 (#201)

- Addition of a `minimum_clearance` function for GEOS >= 3.6.0 (#223)
- Addition of a `offset_curve` function (#229)
- Addition of a `relate_pattern` function (#245)
- Addition of a `clip_by_rect` function (#273)
- Addition of a `reverse` function for GEOS >= 3.7 (#254)
- Addition of `get_precision` to get precision of a geometry and `set_precision` to set the precision of a geometry (may round and reduce coordinates) (#257)

Bug fixes

- Fixed internal GEOS error code detection for `get_dimensions` and `get_srid` (#218)
- Limited the length of geometry repr to 80 characters (#189)
- Fixed error handling in `line_locate_point` for incorrect geometry types, now actually requiring line and point geometries (#216)
- Addition of `get_parts` function to get individual parts of an array of multipart geometries (#197)
- Ensure that `python setup.py clean` removes all previously Cythonized and compiled files (#239)
- Handle GEOS beta versions (#262)

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche
- Mike Taves

Version 0.8 (2020-09-06)

Highlights of this release

- Handle multi geometries in `boundary` (#188)
- Handle empty points in to_wkb by conversion to POINT (nan, nan) (#179)
- Prevent segfault in to_wkt (and repr) with empty points in multipoints (#171)
- Fixed bug in `multilinestrings()`, it now accepts linearrings again (#168)
- Release the GIL to allow for multithreading in functions that do not create geometries (#144) and in the STRtree `query_bulk()` method (#174)
- Addition of a `frechet_distance()` function for GEOS >= 3.7 (#144)
- Addition of `coverage_union()` and `coverage_union_all()` functions for GEOS >= 3.8 (#142)
- Fixed segfaults when adding empty geometries to the STRtree (#147)
- Addition of `include_z=True` keyword in the `get_coordinates()` function to get 3D coordinates (#178)
- Addition of a `build_area()` function for GEOS >= 3.8 (#141)
- Addition of a `normalize()` function (#136)

- Addition of a `make_valid()` function for GEOS >= 3.8 (#107)
- Addition of a `get_z()` function for GEOS >= 3.7 (#175)
- Addition of a `relate()` function (#186)
- The `get_coordinate_dimensions()` function was renamed to `get_coordinate_dimension()` for consistency with GEOS (#176)
- Addition of `covers`, `covered_by`, `contains_properly` predicates to STRtree `query` and `query_bulk` (#157)

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche
- Krishna Chaitanya +
- Martin Fleischmann +
- Tom Clancy +

Version 0.7 (2020-03-18)

Highlights of this release

- STRtree improvements for spatial indexing: * Directly include predicate evaluation in `STRtree.query()` (#87) * Query multiple input geometries (spatial join style) with `STRtree.query_bulk` (#108)
- Addition of a `total_bounds()` function (#107)
- Geometries are now hashable, and can be compared with == or != (#102)
- Fixed bug in `create_collections()` with wrong types (#86)
- Fixed a reference counting bug in STRtree (#97, #100)
- Start of a benchmarking suite using ASV (#96)
- This is the first release that will provide wheels!

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward +
- Casper van der Wel
- Joris Van den Bossche
- Mike Taves +

Version 0.6 (2020-01-31)

Highlights of this release:

- Addition of the STRtree class for spatial indexing (#58)
- Addition of a `bounds` function (#69)
- A new `from_shapely` function to convert Shapely geometries to pygeos.Geometry (#61)
- Reintroduction of the `shared_paths` function (#77)

Contributors:

- Casper van der Wel
- Joris Van den Bossche
- mattijn +

Version 0.5 (2019-10-25)

Highlights of this release:

- Moved to the pygeos GitHub organization.
- Addition of functionality to get and transform all coordinates (eg for reprojections or affine transformations) [#44]
- Ufuncs for converting to and from the WKT and WKB formats [#45]
- `equals_exact` has been added [PR #57]

Version 0.4 (2019-09-16)

This is a major release of PyGEOS and the first one with actual release notes. Most important features of this release are:

- `buffer` and `haussdorff_distance` were completed [#15]
- `voronoi_polygons` and `delaunay_triangles` have been added [#17]
- The PyGEOS documentation is now mostly complete and available on http://pygeos.readthedocs.io .
- The concepts of "empty" and "missing" geometries have been separated. The `pygeos.Empty` and `pygeos.NaG` objects has been removed. Empty geometries are handled the same as normal geometries. Missing geometries are denoted by `None` and are handled by every pygeos function. `NaN` values cannot be used anymore to denote missing geometries. [PR #36]
- Added `pygeos.__version__` and `pygeos.geos_version`. [PR #43]

5.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pygeos.constructive, 70
pygeos.coordinates, 87
pygeos.creation, 27
pygeos.geometry, 14
pygeos.io, 35
pygeos.linear, 84
pygeos.measurement, 40
pygeos.predicates, 46
pygeos.set_operations, 64
pygeos.strtree, 89
pygeos.testing, 94

INDEX

A

apply() (*in module pygeos.coordinates*), 87
area() (*in module pygeos.measurement*), 40
assert_geometries_equal() (*in module pygeos.testing*), 94

B

boundary() (*in module pygeos.constructive*), 70
bounds() (*in module pygeos.measurement*), 41
box() (*in module pygeos.creation*), 27
buffer() (*in module pygeos.constructive*), 71
build_area() (*in module pygeos.constructive*), 72

C

centroid() (*in module pygeos.constructive*), 73
clip_by_rect() (*in module pygeos.constructive*), 73
contains() (*in module pygeos.predicates*), 46
contains_properly() (*in module pygeos.predicates*), 47
convex_hull() (*in module pygeos.constructive*), 74
count_coordinates() (*in module pygeos.coordinates*), 87
coverage_union() (*in module pygeos.set_operations*), 64
coverage_union_all() (*in module pygeos.set_operations*), 64
covered_by() (*in module pygeos.predicates*), 47
covers() (*in module pygeos.predicates*), 48
crosses() (*in module pygeos.predicates*), 49

D

delaunay_triangles() (*in module pygeos.constructive*), 74
destroy_prepared() (*in module pygeos.creation*), 28
difference() (*in module pygeos.set_operations*), 65
disjoint() (*in module pygeos.predicates*), 50
distance() (*in module pygeos.measurement*), 41
dwithin() (*in module pygeos.predicates*), 51

E

empty() (*in module pygeos.creation*), 28
envelope() (*in module pygeos.constructive*), 75
equals() (*in module pygeos.predicates*), 52
equals_exact() (*in module pygeos.predicates*), 52
extract_unique_points() (*in module pygeos.constructive*), 75

F

force_2d() (*in module pygeos.geometry*), 14
force_3d() (*in module pygeos.geometry*), 15
frechet_distance() (*in module pygeos.measurement*), 42
from_geojson() (*in module pygeos.io*), 35
from_shapely() (*in module pygeos.io*), 36
from_wkb() (*in module pygeos.io*), 36
from_wkt() (*in module pygeos.io*), 37

G

geometrycollections() (*in module pygeos.creation*), 29
get_coordinate_dimension() (*in module pygeos.geometry*), 15
get_coordinates() (*in module pygeos.coordinates*), 88
get_dimensions() (*in module pygeos.geometry*), 16
get_exterior_ring() (*in module pygeos.geometry*), 16
get_geometry() (*in module pygeos.geometry*), 17
get_interior_ring() (*in module pygeos.geometry*), 17
get_num_coordinates() (*in module pygeos.geometry*), 18
get_num_geometries() (*in module pygeos.geometry*), 18
get_num_interior_rings() (*in module pygeos.geometry*), 19
get_num_points() (*in module pygeos.geometry*), 19
get_parts() (*in module pygeos.geometry*), 20
get_point() (*in module pygeos.geometry*), 21
get_precision() (*in module pygeos.geometry*), 21
get_rings() (*in module pygeos.geometry*), 22
get_srid() (*in module pygeos.geometry*), 23
get_type_id() (*in module pygeos.geometry*), 23
get_x() (*in module pygeos.geometry*), 24
get_y() (*in module pygeos.geometry*), 24
get_z() (*in module pygeos.geometry*), 25

H

has_z() (*in module pygeos.predicates*), 53
hausdorff_distance() (*in module pygeos.measurement*), 42

I

intersection() (*in module pygeos.set_operations*), 66
intersection_all() (*in module pygeos.set_operations*), 67
intersects() (*in module pygeos.predicates*), 53
is_ccw() (*in module pygeos.predicates*), 54
is_closed() (*in module pygeos.predicates*), 55
is_empty() (*in module pygeos.predicates*), 55
is_geometry() (*in module pygeos.predicates*), 56
is_missing() (*in module pygeos.predicates*), 56
is_prepared() (*in module pygeos.predicates*), 57
is_ring() (*in module pygeos.predicates*), 57
is_simple() (*in module pygeos.predicates*), 58
is_valid() (*in module pygeos.predicates*), 59
is_valid_input() (*in module pygeos.predicates*), 59
is_valid_reason() (*in module pygeos.predicates*), 60

L

length() (*in module pygeos.measurement*), 43
line_interpolate_point() (*in module pygeos.linear*), 84
line_locate_point() (*in module pygeos.linear*), 85
line_merge() (*in module pygeos.linear*), 85
linearrings() (*in module pygeos.creation*), 29
linestrings() (*in module pygeos.creation*), 30

M

make_valid() (*in module pygeos.constructive*), 76
minimum_bounding_circle() (*in module pygeos.constructive*), 76
minimum_bounding_radius() (*in module pygeos.measurement*), 43
minimum_clearance() (*in module pygeos.measurement*), 44
module
    pygeos.constructive, 70
    pygeos.coordinates, 87
    pygeos.creation, 27
    pygeos.geometry, 14
    pygeos.io, 35
    pygeos.linear, 84
    pygeos.measurement, 40
    pygeos.predicates, 46
    pygeos.set_operations, 64
    pygeos.strtree, 89
    pygeos.testing, 94
multilinestrings() (*in module pygeos.creation*), 31
multipoints() (*in module pygeos.creation*), 31
multipolygons() (*in module pygeos.creation*), 32

N

nearest() (*STRtree method*), 90
nearest_all() (*STRtree method*), 90
normalize() (*in module pygeos.constructive*), 77

O

offset_curve() (*in module pygeos.constructive*), 77
oriented_envelope() (*in module pygeos.constructive*), 78
overlaps() (*in module pygeos.predicates*), 60

P

point_on_surface() (*in module pygeos.constructive*), 78
points() (*in module pygeos.creation*), 33
polygonize() (*in module pygeos.constructive*), 79
polygonize_full() (*in module pygeos.constructive*), 80
polygons() (*in module pygeos.creation*), 33
prepare() (*in module pygeos.creation*), 34
pygeos.constructive
    module, 70
pygeos.coordinates
    module, 87
pygeos.creation
    module, 27
pygeos.geometry
    module, 14
pygeos.io
    module, 35
pygeos.linear
    module, 84
pygeos.measurement
    module, 40
pygeos.predicates
    module, 46
pygeos.set_operations
    module, 64
pygeos.strtree
    module, 89
pygeos.testing
    module, 94

Q

query() (*STRtree method*), 92
query_bulk() (*STRtree method*), 92

R

relate() (*in module pygeos.predicates*), 61
relate_pattern() (*in module pygeos.predicates*), 61
reverse() (*in module pygeos.constructive*), 81

S

segmentize() (*in module pygeos.constructive*), 81
set_coordinates() (*in module pygeos.coordinates*), 88
set_precision() (*in module pygeos.geometry*), 25
set_srid() (*in module pygeos.geometry*), 27
shared_paths() (*in module pygeos.linear*), 86
shortest_line() (*in module pygeos.linear*), 86
simplify() (*in module pygeos.constructive*), 82
snap() (*in module pygeos.constructive*), 82
STRtree (*class in pygeos.strtree*), 89
symmetric_difference() (*in module pygeos.set_operations*), 67
symmetric_difference_all() (*in module pygeos.set_operations*), 68

T

to_geojson() (*in module pygeos.io*), 37
to_shapely() (*in module pygeos.io*), 38
to_wkb() (*in module pygeos.io*), 38
to_wkt() (*in module pygeos.io*), 39
total_bounds() (*in module pygeos.measurement*), 45
touches() (*in module pygeos.predicates*), 62

U

union() (*in module pygeos.set_operations*), 68
union_all() (*in module pygeos.set_operations*), 69

V

voronoi_polygons() (*in module pygeos.constructive*), 83

W

within() (*in module pygeos.predicates*), 63