pygeos

Casper van der Wel

Mar 04, 2021

CONTENTS:

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.

Note: PyGEOS is a very young package. While the available functionality should be stable and working correctly, it's still possible that APIs change in upcoming releases. But we would love for you to try it out, give feedback or contribute!

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

RELATIONSHIP TO SHAPELY

Both Shapely and PyGEOS are exposing the functionality of the GEOS C++ library to Python. While Shapely only deals with single geometries, PyGEOS provides vectorized functions to work with arrays of geometries, giving better performance and convenience for such usecases.

There is active discussion and work toward integrating PyGEOS into Shapely:

- latest proposal: https://github.com/shapely/shapely-rfc/pull/1
- prior discussion: https://github.com/Toblerity/Shapely/issues/782

For now PyGEOS is developed as a separate project.

CHAPTER

FIVE

REFERENCES

- GEOS: http://trac.osgeo.org/geos
- 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

SIX

COPYRIGHT & LICENSE

6.1 API Reference

6.1.1 Installation

Installation from PyPI

PyGEOS is available as a binary distribution (wheel) for Linux, OSX and Windows platforms. Install as follows:

```
$ pip install pygeos
```

Installation using conda

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

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

Installation with custom GEOS libary

On Linux:

```
$ sudo apt install libgeos-dev
```

On OSX:

```
$ brew install geos
```

Make sure *geos-config* is available from you shell; append PATH if necessary:

```
$ export PATH=$PATH:/path/to/dir/having/geos-config
$ 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
- Run `pip install pygeos --no-binary`

Installation from source

The same as above, but then instead of installing pygeos with pip, you clone the package from Github:

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

Install Cython, which is required to build Cython extensions:

```
$ pip install cython
```

Install it in development mode using *pip*:

```
$ pip install -e .[test] --no-build-isolation
```

Run the unittests:

```
$ pytest pygeos
```

Notes on GEOS discovery

If GEOS is installed, normally the `geos-config` command line utility will be available, and `pip install` will find GEOS automatically. But if needed, you can specify where PyGEOS should look for the GEOS library before installing it:

On Linux / OSX:

```
$ export GEOS_INCLUDE_PATH=$CONDA_PREFIX/Library/include
$ export GEOS_LIBRARY_PATH=$CONDA_PREFIX/Library/lib
```

On Windows (assuming you are in a Visual C++ shell):

```
$ set GEOS_INCLUDE_PATH=%CONDA_PREFIX%\Library\include
$ set GEOS_LIBRARY_PATH=%CONDA_PREFIX%\Library\lib
```

6.1.2 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 and it may convert empty points. 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. Also, different kind of empty points will be considered equal. 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.

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

Returns -1 for not-a-geometry values.

Parameters

geometry [Geometry or array_like]

Examples

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

pygeos.geometry.**get_dimensions**(*geometry*)
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 and None geometries return -1.

Parameters

geometry [Geometry or array_like]

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
```

pygeos.geometry.**get_exterior_ring**(*geometry*)
Returns the exterior ring of a polygon.

Parameters

geometry [Geometry or array_like]

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
```

pygeos.geometry.**get_geometry**(*geometry*, *index*)

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.

See also:

get_num_geometries

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
```

pygeos.geometry.**get_interior_ring**(*geometry*, *index*)

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.

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
```

pygeos.geometry.**get_num_coordinates**(*geometry*)

Returns the total number of coordinates in a geometry.

Returns 0 for not-a-geometry values.

Parameters

geometry [Geometry or array_like]

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
```

pygeos.geometry.**get_num_geometries**(*geometry*)

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.

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
```

pygeos.geometry.**get_num_interior_rings**(*geometry*)

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.

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
```

pygeos.geometry.**get_num_points**(*geometry*)

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.

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
```

pygeos.geometry.**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, optional (default: False)] If True, will return a tuple of ndarrys 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)

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]
```

pygeos.geometry.**get_point**(*geometry*, *index*)

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.

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
```

pygeos.geometry.**get_precision**(*geometry*)

Get the precision of a geometry.

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]

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
```

pygeos.geometry.**get_srid**(*geometry*)

Returns the SRID of a geometry.

Returns -1 for not-a-geometry values.

Parameters

geometry [Geometry or array_like]

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
```

pygeos.geometry.**get_type_id**(*geometry*)

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]

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]
```

pygeos.geometry.**get_x**(*point*)

Returns the x-coordinate of a point

Parameters

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

See also:

get_y, ***get_z***

Examples

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

pygeos.geometry.**get_y**(*point*)

Returns the y-coordinate of a point

Parameters

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

See also:

get_x, ***get_z***

Examples

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

pygeos.geometry.**get_z**(*point*)

Returns the z-coordinate of a point.

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.

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
```

pygeos.geometry.**set_precision**(*geometry*, *grid_size*, *preserve_topology=False*)

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

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

Coordinates will be rounded if a precision grid is less precise than the input geometry. Duplicated vertices will be dropped from lines and polygons for grid sizes greater than 0. Line and polygon geometries may collapse to empty geometries if all vertices are closer together than grid_size. 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.

preserve_topology [bool, optional (default: False)] If True, will attempt to preserve the topology of a geometry after rounding coordinates.

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(None, 1.0) is None
True
```

pygeos.geometry.**set_srid**(*geometry*, *srid*)

Returns a geometry with its SRID set.

Parameters

geometry [Geometry or array_like]

srid [int]

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
```

6.1.3 Geometry creation

pygeos.creation.**box**(*x1*, *y1*, *x2*, *y2*)

Create box polygons.

Parameters

x1 [array_like]

y1 [array_like]

x2 [array_like]

y2 [array_like]

pygeos.creation.**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

See also:

prepare

pygeos.creation.**geometrycollections**(*geometries*)
Create geometrycollections from arrays of geometries

Parameters

geometries [array_like] An array of geometries

pygeos.creation.**linearrings**(*coords*, *y=None*, *z=None*)
Create an array of linearrings.

If the provided coords do not constitute a closed linestring, the first coordinate is duplicated at the end to close the ring.

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]

z [array_like]

pygeos.creation.**linestrings**(*coords*, *y=None*, *z=None*)
Create an array of linestrings.

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]

z [array_like]

pygeos.creation.**multilinestrings**(*geometries*)
Create multilinestrings from arrays of linestrings

Parameters

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

pygeos.creation.**multipoints**(*geometries*)
Create multipoints from arrays of points

Parameters

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

pygeos.creation.**multipolygons**(*geometries*)
Create multipolygons from arrays of polygons

Parameters

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

pygeos.creation.**points**(*coords*, *y=None*, *z=None*)
Create an array of points.

Note that GEOS >=3.10 automatically converts POINT (nan nan) to POINT EMPTY.

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]

z [array_like]

pygeos.creation.**polygons**(*shells*, *holes=None*)
Create an array of polygons.

Parameters

shell [array_like] An array of linearrings that constitute the out shell of the polygons. Coordinates can also be passed, see linearrings.

holes [array_like] An array of lists of linearrings that constitute holes for each shell.

pygeos.creation.**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

See also:

is_prepared Identify whether a geometry is prepared already.

destroy_prepared Destroy the prepared part of a geometry.

6.1.4 Input/Output

pygeos.io.**from_shapely**(*geometry*, ***kwargs*)
Creates geometries from shapely Geometry objects.

This function requires the GEOS version of PyGEOS and shapely to be equal.

Parameters

geometry [shapely Geometry object or array_like]

Examples

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

pygeos.io.**from_wkb**(*geometry*, ***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.

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)>
```

pygeos.io.**from_wkt**(*geometry*, ***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.

Examples

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

pygeos.io.**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
- empty points are transformed to 3D in GEOS < 3.8
- empty points are transformed to 2D in GEOS 3.8

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

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

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'
```

pygeos.io.**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.

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] Whether to 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)".

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)'
```

(continues on next page)

(continued from previous page)

```
>>> 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)'
```

6.1.5 Measurement

pygeos.measurement.**area**(*geometry*, ***kwargs*)

Computes the area of a (multi)polygon.

Parameters

geometry [Geometry or array_like]

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
```

pygeos.measurement.**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]

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]
```

pygeos.measurement.**distance**(*a*, *b*, ***kwargs*)

Computes the Cartesian distance between two geometries.

Parameters

a, b [Geometry or array_like]

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
```

pygeos.measurement.**frechet_distance**(*a*, *b*, *densify=None*, ***kwargs*)

Compute the discrete Fréchet distance between two geometries.

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, array_like or None] The value of densify is required to be between 0 and 1.

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
```

pygeos.measurement.**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, array_like or None] The value of densify is required to be between 0 and 1.

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
```

pygeos.measurement.**length**(*geometry*, ***kwargs*)

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

Parameters

geometry [Geometry or array_like]

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
```

pygeos.measurement.**minimum_clearance**(*geometry*, ***kwargs*)

Computes the Minimum Clearance distance.

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]

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
```

pygeos.measurement.**total_bounds**(*geometry*, ***kwargs*)

Computes the total bounds (extent) of the geometry.

Parameters

geometry [Geometry or array_like]

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]

6.1.6 Predicates

pygeos.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]

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

pygeos.predicates.**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]

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
```

pygeos.predicates.**covered_by**(*a*, *b*, ***kwargs*)

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

Parameters

a, b [Geometry or array_like]

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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

pygeos.predicates.**covers**(*a*, *b*, ***kwargs*)

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

Parameters

a, b [Geometry or array_like]

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
```

pygeos.predicates.**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]

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
```

pygeos.predicates.**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]

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
```

pygeos.predicates.**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]

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
```

pygeos.predicates.**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]

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
```

pygeos.predicates.**has_z**(*geometry*, ***kwargs*)

Returns True if a geometry has a Z coordinate.

Parameters

geometry [Geometry or array_like]

Examples

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

pygeos.predicates.**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]

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
```

pygeos.predicates.**is_ccw**(*geometry*, ***kwargs*)

Returns True if a linestring or linearring is counterclockwise.

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).

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
```

pygeos.predicates.**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.

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
```

pygeos.predicates.**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.

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
```

pygeos.predicates.**is_geometry**(*geometry*, ***kwargs*)

Returns True if the object is a geometry

Parameters

geometry [any object or array_like]

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
```

pygeos.predicates.**is_missing**(*geometry*, ***kwargs*)

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

Parameters

geometry [any object or array_like]

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
```

pygeos.predicates.**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]

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
```

pygeos.predicates.**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.

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)
```

pygeos.predicates.**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.

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
```

pygeos.predicates.**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.

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
```

pygeos.predicates.**is_valid_input**(*geometry*, ***kwargs*)

Returns True if the object is a geometry or None

Parameters

geometry [any object or array_like]

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
```

pygeos.predicates.**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.

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))"))
'Self-intersection[0 0]'
>>> is_valid_reason(None) is None
True
```

pygeos.predicates.**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]

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
```

pygeos.predicates.**relate**(*a*, *b*, ***kwargs*)

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

Parameters

a, b [Geometry or array_like]

Examples

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

pygeos.predicates.**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]

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
```

pygeos.predicates.**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]

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)"))
```

(continues on next page)

(continued from previous page)

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

pygeos.predicates.**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]

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

6.1.7 Set operations

pygeos.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.

Requires at least GEOS 3.8.0.

Parameters

a [Geometry or array_like]

b [Geometry or array_like]

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))>

pygeos.set_operations.**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.

Requires at least GEOS 3.8.0.

Parameters

geometries [array_like]

axis [int (default None)] 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.

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))>
```

pygeos.set_operations.**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 (default: None).] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

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))>
```

pygeos.set_operations.**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 (default: None).] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

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))
<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))>
```

pygeos.set_operations.**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 (default None)] 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.

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)>]
```

pygeos.set_operations.**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 (default: None).] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

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,...>
```

pygeos.set_operations.**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 (default None)] 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.

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))>]
```

pygeos.set_operations.**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 (default: None).] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

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))>
```

pygeos.set_operations.**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 (default: None).] Precision grid size; requires GEOS >= 3.9.0. Will use the highest precision of the inputs by default.

axis [int (default None)] 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.

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))>
```

6.1.8 Constructive operations

pygeos.constructive.**boundary**(*geometry*, ***kwargs*)

Returns the topological boundary of a geometry.

Parameters

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

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
```

pygeos.constructive.**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] Specifies the number of linear segments in a quarter circle in the approximation of circular arcs.

cap_style [{'round', 'square', 'flat'}] 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'}] 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] Crops of 'mitre'-style joins if the point is displaced from the buffered vertex by more than this limit.

single_sided [bool] Only buffer at one side of the geometry.

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, 11.4 8.59, 10 8, 8.59 8.59, 8 10, 8.59 11....>
>>> 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, 21.8 9, 21 8.17, 10 8, 10 12...>
>>> 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
```

`pygeos.constructive.`**`build_area`**(*geometry*, ***kwargs*)

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

Equivalent of the PostGIS ST_BuildArea() function.

Requires at least GEOS 3.8.0.

Parameters

geometry [Geometry or array_like]

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))>
```

pygeos.constructive.**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]

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>
```

pygeos.constructive.**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

Examples

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

pygeos.constructive.**convex_hull**(*geometry*, ***kwargs*)

Computes the minimum convex geometry that encloses an input geometry.

Parameters

geometry [Geometry or array_like]

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>
```

pygeos.constructive.**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] Snap input vertices together if their distance is less than this value.

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

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>
```

pygeos.constructive.**envelope**(*geometry*, ***kwargs*)

Computes the minimum bounding box that encloses an input geometry.

Parameters

geometry [Geometry or array_like]

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>
```

pygeos.constructive.**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]

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>
```

pygeos.constructive.**make_valid**(*geometry*, ***kwargs*)

Repairs invalid geometries.

Requires at least GEOS 3.8.0.

Parameters

geometry [Geometry or array_like]

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))>
```

pygeos.constructive.**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]

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))>
```

pygeos.constructive.**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] Specifies the number of linear segments in a quarter circle in the approximation of circular arcs.

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

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)>
```

pygeos.constructive.**point_on_surface**(*geometry*, ***kwargs*)

Returns a point that intersects an input geometry.

Parameters

geometry [Geometry or array_like]

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>
```

pygeos.constructive.**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.

Returns

GeometryCollection or array of GeometryCollections

See also:

`get_parts`, `get_geometry`

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)))>
```

pygeos.constructive.**reverse**(*geometry*, ***kwargs*)

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

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]

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
```

pygeos.constructive.**segmentize**(*geometry*, *tolerance*, ***kwargs*)

'segmentize' requires at least GEOS 3.10.0

pygeos.constructive.**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] If set to True, the operation will avoid creating invalid geometries.

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))>
```

pygeos.constructive.**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]

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))>
```

pygeos.constructive.**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] Snap input vertices together if their distance is less than this value.

extend_to [Geometry or array_like] 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] If set to True, the triangulation will return a collection of linestrings instead of polygons.

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>
```

6.1.9 Linestring operations

pygeos.linear.**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] If normalized is set to True, the distance is a fraction of the total line length instead of the absolute distance.

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

pygeos.linear.**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] If normalized is set to True, the distance is a fraction of the total line length instead of the absolute distance.

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
```

pygeos.linear.**line_merge**(*line*)
Returns (multi)linestrings formed by combining the lines in a multilinestrings.

Parameters

line [Geometry or array_like]

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>
```

pygeos.linear.**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]

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 ...>
```

6.1.10 Coordinate operations

pygeos.coordinates.**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] Whether to include the third dimension in the coordinates array that is passed to the *transformation* function. If True, and 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)>
```

pygeos.coordinates.**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
```

pygeos.coordinates.**get_coordinates**(*geometry*, *include_z=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] Whether to include the third dimension in the output. If True, and a geometry has no third dimension, the z-coordinates will be NaN.

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]]
```

pygeos.coordinates.**set_coordinates**(*geometry*, *coordinates*)

Returns a copy of a geometry array with different coordinates.

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.

Parameters

geometry [Geometry or array_like]

coordinates: array_like

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)>
```

6.1.11 STRTree

class pygeos.strtree.**STRtree**(*geometries*, *leafsize=10*)

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

For two-dimensional spatial data. The actual tree will be constructed at the first query.

Parameters

geometries [array_like]

leafsize [int] 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.

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.

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 (default: None)] Maximum distance within which to query for nearest items in tree. Must be greater than 0.

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

query(*geometry*, *predicate=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).

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'}, optional] The predicate to use for testing geometries from the tree that are within the input geometry's envelope.

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_bulk(*geometry*, *predicate=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).

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'}, optional] The predicate to use for testing geometries from the tree that are within the input geometry's envelope.

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:
```

(continues on next page)

(continued from previous page)

```
>>> 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]]
```

6.1.12 Changelog

Version 0.10 (unreleased)

Major enhancements

- Addition of `nearest` and `nearest_all` functions to `STRtree` for GEOS >= 3.6 to find the nearest neighbors (#272).

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.

Added GEOS functions

- Addition of a `contains_properly` function (#267)
- Addition of a `polygonize` function (#275)
- Addition of a `segmentize` function for GEOS >= 3.10 (#299)

Bug fixes

- Fixed portability issue for ARM architecture (#293)

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 0.9 (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]

6.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

T

U

V

W