Shapely Documentation

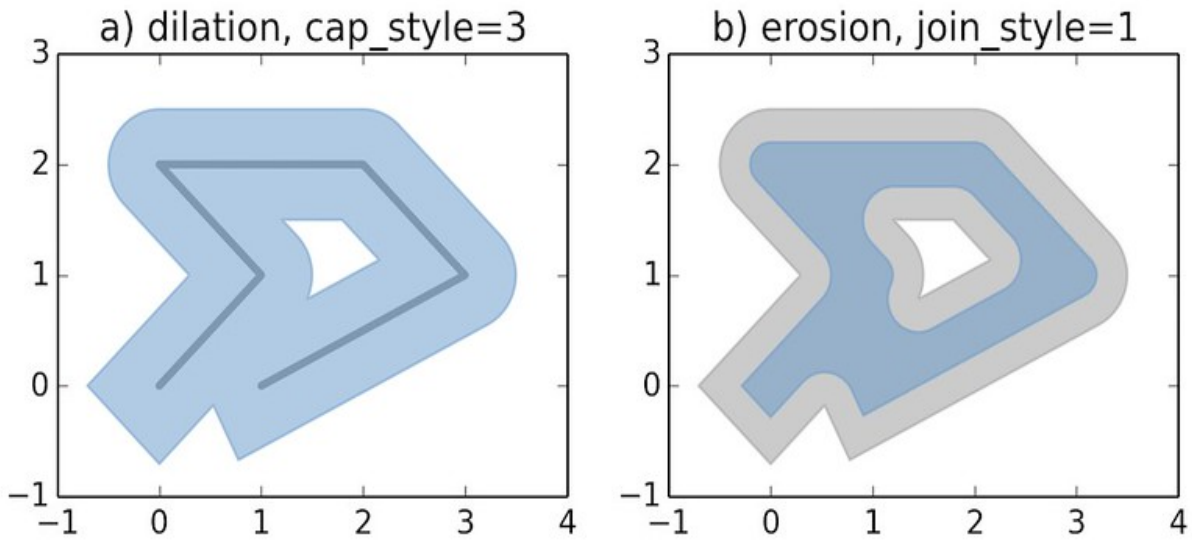
Release 2.0.3

Sean Gillies

Apr 16, 2024

USER GUIDE

Manipulation and analysis of geometric objects in the Cartesian plane.



Shapely is a BSD-licensed Python package for manipulation and analysis of planar geometric objects. It is using the widely deployed open-source geometry library GEOS (the engine of PostGIS, and a port of JTS). Shapely wraps GEOS geometries and operations to provide both a feature rich *Geometry* interface for singular (scalar) geometries and higher-performance NumPy ufuncs for operations using arrays of geometries. Shapely is not primarily focused on data serialization formats or coordinate systems, but can be readily integrated with packages that are.

CHAPTER

ONE

WHAT IS A UFUNC?

A universal function (or ufunc for short) is a function that operates on *n*-dimensional arrays on an element-by-element fashion and supports array broadcasting. The underlying `for` loops are implemented in C to reduce the overhead of the Python interpreter.

CHAPTER

TWO

MULTITHREADING

Shapely functions generally support multithreading by releasing the Global Interpreter Lock (GIL) during execution. Normally in Python, the GIL prevents multiple threads from computing at the same time. Shapely functions internally release this constraint so that the heavy lifting done by GEOS can be done in parallel, from a single Python process.

2.1 Usage

Here is the canonical example of building an approximately circular patch by buffering a point, using the scalar Geometry interface:

```
>>> from shapely import Point
>>> patch = Point(0.0, 0.0).buffer(10.0)
>>> patch
<POLYGON ((10 0, 9.952 -0.98, 9.808 -1.951, 9.569 -2.903, 9.239 -3.827, 8.81...>
>>> patch.area
313.6548490545941
```

Using the vectorized ufunc interface (instead of using a manual for loop), compare an array of points with a polygon:

```
>>> import shapely
>>> import numpy as np
>>> geoms = np.array([Point(0, 0), Point(1, 1), Point(2, 2)])
>>> polygon = shapely.box(0, 0, 2, 2)

>>> shapely.contains(polygon, geoms)
array([False,  True, False])
```

See the documentation for more examples and guidance: https://shapely.readthedocs.io

2.2 Requirements

Shapely 2.0 requires

- Python >=3.7
- GEOS >=3.5
- NumPy >=1.14

2.3 Installing Shapely

We recommend installing Shapely using one of the available built distributions, for example using `pip` or `conda`:

```
$ pip install shapely
# or using conda
$ conda install shapely --channel conda-forge
```

See the installation documentation for more details and advanced installation instructions.

2.4 Integration

Shapely does not read or write data files, but it can serialize and deserialize using several well known formats and protocols. The shapely.wkb and shapely.wkt modules provide dumpers and loaders inspired by Python's pickle module.

```
>>> from shapely.wkt import dumps, loads
>>> dumps(loads('POINT (0 0)'))
'POINT (0.0000000000000000 0.0000000000000000)'
```

Shapely can also integrate with other Python GIS packages using GeoJSON-like dicts.

```
>>> import json
>>> from shapely.geometry import mapping, shape
>>> s = shape(json.loads('{"type": "Point", "coordinates": [0.0, 0.0]}'))
>>> s
<POINT (0 0)>
>>> print(json.dumps(mapping(s)))
{"type": "Point", "coordinates": [0.0, 0.0]}
```

2.5 Support

Questions about using Shapely may be asked on the GIS StackExchange using the "shapely" tag.

Bugs may be reported at https://github.com/shapely/shapely/issues.

2.6 Copyright & License

Shapely is licensed under BSD 3-Clause license. GEOS is available under the terms of GNU Lesser General Public License (LGPL) 2.1 at https://libgeos.org.

2.7 Credits

Shapely is written by:

- Adi Shavit <adishavit@gmail.com>
- Alan D. Snow <alansnow21@gmail.com>
- Alberto Rubiales <arubiales11@gmail.com>
- Allan Adair <allan.m.adair@gmail.com>
- Andrew Blakey <ablakey@gmail.com>
- Andy Freeland <andy@andyfreeland.net>
- Ariel Kadouri <ariel@arielsartistry.com>
- Aron Bierbaum <aronbierbaum@gmail.com>
- Bart Broere <2715782+bartbroere@users.noreply.github.com>
- Bas Couwenberg <sebastic@xs4all.nl>
- Ben Beasley <code@musicinmybrain.net>
- Benjamin Root <ben.v.root@gmail.com>
- BertrandGervais <bertrand.gervais.pro@gmail.com>
- Bhavika Tekwani <4955119+bhavika@users.noreply.github.com>
- Bi0T1N <Bi0T1N@users.noreply.github.com>
- Brad Hards <bradh@frogmouth.net>
- Brendan Ward <bcward@astutespruce.com>
- Brandon Wood <btwood@geometeor.com>
- Casper van der Wel <caspervdw@gmail.com>
- Chad Hawkins <cwh@chadwhawkins.com>
- Christian Prior <cprior@gmail.com>
- Christian Quest <github@cquest.org>
- Christophe Pradal <christophe.pradal@inria.fr>
- Dan Baston <dbaston@gmail.com>
- Dan Mahr <danmahr23@gmail.com>
- Daniele Esposti <expobrain@users.noreply.github.com>
- Dave Collins <dave@hopest.net>
- David Baumgold <david@davidbaumgold.com>
- David Swinkels <davidswinkelss@gmail.com>
- Denis Rykov <rykovd@gmail.com>
- Enrico Ferreguti <enricofer@gmail.com>
- Erwin Sterrenburg <e.w.sterrenburg@gmail.com>
- Ewout ter Hoeven <E.M.terHoeven@student.tudelft.nl>

- Felix Divo <4403130+felixdivo@users.noreply.github.com>
- Felix Yan <felixonmars@archlinux.org>
- Filipe Fernandes <ocefpaf@gmail.com>
- Frédéric Junod <frederic.junod@camptocamp.com>
- Gabi Davar <grizzly.nyo@gmail.com>
- Gerrit Holl <gerrit.holl@dwd.de>
- Hannes <kannes@users.noreply.github.com>
- Hao Zheng <Furioushaozheng@gmail.com>
- Henry Walshaw <henry.walshaw@gmail.com>
- Howard Butler <hobu.inc@gmail.com>
- Hugo <hugovk@users.noreply.github.com>
- Idan Miara <idan@miara.com>
- Jacob Wasserman <jwasserman@gmail.com>
- Jaeha Lee <jaehaaheaj@gmail.com>
- James Douglass <jamesdouglassusa@gmail.com>
- James Gaboardi <jgaboardi@gmail.com>
- James Lamb <jaylamb20@gmail.com>
- James McBride <jdmcbr@gmail.com>
- James Spencer <james.s.spencer@gmail.com>
- Jamie Hall <jamie1212@gmail.com>
- Jason Sanford <jason.sanford@mapmyfitness.com>
- Jeethu Rao <jeethu@jeethurao.com>
- Jeremiah England <34973839+Jeremiah-England@users.noreply.github.com>
- Jinkun Wang <mejkunw@gmail.com>
- Johan Euphrosine <proppy@aminche.com>
- Johannes Schönberger <jschoenberger@demuc.de>
- Jonathan Schoonhoven <jschoonhoven@lyft.com>
- Joris Van den Bossche <jorisvandenbossche@gmail.com>
- Joshua Arnott <josh@snorfalorpagus.net>
- Juan Luis Cano Rodríguez <juanlu@satellogic.com>
- Justin Shenk <shenk.justin@gmail.com>
- Kai Lautaportti <dokai@b426a367-1105-0410-b9ff-cdf4ab011145>
- Kelsey Jordahl <kjordahl@enthought.com>
- Kevin Wurster <wursterk@gmail.com>
- Konstantin Veretennicov <kveretennicov@gmail.com>
- Koshy Thomas <koshy1123@gmail.com>

- Krishna Chaitanya <bkchaitan94@gmail.com>
- Kristian Evers <kristianevers@gmail.com>
- Kyle Barron <kylebarron2@gmail.com>
- Leandro Lima <leandro@limaesilva.com.br>
- Lukasz <uhho@users.noreply.github.com>
- Luke Lee <durdenmisc@gmail.com>
- Maarten Vermeyen <maarten.vermeyen@rwo.vlaanderen.be>
- Marc Jansen <jansen@terrestris.de>
- Marco De Nadai <me@marcodena.it>
- Martin Fleischmann <martin@martinfleischmann.net>
- Mathieu <mathieu.nivel@gmail.com>
- Matt Amos <matt.amos@mapzen.com>
- Matthias Cuntz <mcuntz@users.noreply.github.com>
- MejstrikRudolf <68251685+MejstrikRudolf@users.noreply.github.com>
- Michael K <michael-k@users.noreply.github.com>
- Michel Blancard <michel.blancard@data.gouv.fr>
- Mike Taves <mwtoews@gmail.com>
- Morris Tweed <tweed.morris@gmail.com>
- Naveen Michaud-Agrawal <naveen.michaudagrawal@gmail.com>
- Oliver Tonnhofer <olt@bogosoft.com>
- Paveł Tyślacki <tbicr@users.noreply.github.com>
- Peter Sagerson <psagers.github@ignorare.net>
- Phil Elson <pelson.pub@gmail.com>
- Pierre PACI <villerupt@gmail.com>
- Raja Gangopadhya <raja.gangopadhya@ridewithvia.com>
- Ricardo Zilleruelo <51384295+zetaatlyft@users.noreply.github.com>
- Rémy Phelipot <remy-phelipot@users.noreply.github.com>
- S Murthy <sr-murthy@users.noreply.github.com>
- Sampo Syrjanen <sampo.syrjanen@here.com>
- Samuel Chin <samuelchin91@gmail.com>
- Sean Gillies <sean.gillies@gmail.com>
- Sobolev Nikita <mail@sobolevn.me>
- Stephan Hügel <urschrei@gmail.com>
- Steve M. Kim <steve@climate.com>
- Taro Matsuzawa aka. btm <btm@tech.email.ne.jp>
- Thibault Deutsch <thibault.deutsch@gmail.com>

- Thomas Gratier <thomas_gratier@yahoo.fr>
- Thomas Kluyver <takowl@gmail.com>
- Tim Gates <tim.gates@iress.com>
- Tobias Sauerwein <tobias.sauerwein@camptocamp.com>
- Tom Caruso <carusot42@gmail.com>
- Tom Clancy <17627475+clncy@users.noreply.github.com>
- WANG Aiyong <gepcelway@gmail.com>
- Will May <williamcmay@live.com>
- Zachary Ware <zachary.ware@gmail.com>
- aharfoot <aharfoot@users.noreply.github.com>
- bstadlbauer <11799671+bstadlbauer@users.noreply.github.com>
- cclauss <cclauss@me.com>
- clefrks <33859587+clefrks@users.noreply.github.com>
- davidh-ssec <david.hoese@ssec.wisc.edu>
- georgeouzou <geothrock@gmail.com>
- giumas <gmasetti@ccom.unh.edu>
- gpapadok <38889721+gpapadok@users.noreply.github.com>
- joelostblom <joelostblom@users.noreply.github.com>
- ljwolf <levi.john.wolf@gmail.com>
- mindw <grizzly.nyo@gmail.com>
- rsmb <rsmb@users.noreply.github.com>
- shongololo <garethsimons@me.com>
- solarjoe <walterwhite666@googlemail.com>
- sshuair <sshuair@gmail.com>
- stephenworsley <49274989+stephenworsley@users.noreply.github.com>

See also: https://github.com/shapely/shapely/graphs/contributors.

Additional help from:

- Justin Bronn (GeoDjango) for ctypes inspiration
- Martin Davis (JTS)
- Sandro Santilli, Mateusz Loskot, Paul Ramsey, et al (GEOS Project)

Major portions of this work were supported by a grant (for Pleiades) from the U.S. National Endowment for the Humanities (https://www.neh.gov).

2.8 Frequently asked questions and answers

CHAPTER

THREE

I INSTALLED SHAPELY IN A CONDA ENVIRONMENT USING PIP. WHY DOESN'T IT WORK?

Shapely versions < 2.0 load a GEOS shared library using ctypes. It's not uncommon for users to have multiple copies of GEOS libs on their system. Loading the correct one is complicated and shapely has a number of platform-dependent GEOS library loading bugs. The project has particularly poor support for finding the correct GEOS library for a shapely package installed from PyPI *into* a conda environment. We recommend that conda users always get shapely from conda-forge.

CHAPTER

FOUR

ARE THERE REFERENCES FOR THE ALGORITHMS USED BY SHAPELY?

Generally speaking, shapely's predicates and operations are derived from methods of the same name from GEOS and the JTS Topology Suite. See the JTS FAQ for references describing the JTS algorithms.

CHAPTER

FIVE

I USED .BUFFER() ON A GEOMETRY WITH Z COORDINATES. WHERE DID THE Z COORDINATES GO?

The buffer algorithm in GEOS is purely two-dimensional and discards any Z coordinates. This is generally the case for the GEOS algorithms.

5.1 Installation

5.1.1 Built distributions

Built distributions don't require compiling Shapely and its dependencies, and can be installed using `pip` or `conda`. In addition, Shapely is also available via some system package management tools like apt.

Installation from PyPI

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

```
$ pip install shapely
```

Installation using conda

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

```
$ conda install shapely --channel conda-forge
```

5.1.2 Installation from source with custom GEOS libary

You may want to use a specific GEOS version or a GEOS distribution that is already present on your system (for compatibility with other modules that depend on GEOS, such as cartopy or osgeo.ogr). In such cases you will need to ensure the GEOS library is installed on your system and then compile Shapely from source yourself, by directing pip to ignore the binary wheels.

On Linux:

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

On macOS:

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

If you've installed GEOS to a standard location on Linux or macOS, the installation will automatically find it using `geos-config`. See the notes below on GEOS discovery at compile time to configure this.

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

- Get a C compiler applicable to your Python version (https://wiki.python.org/moin/WindowsCompilers)
- Download and install a GEOS binary (https://trac.osgeo.org/osgeo4w/)
- Set GEOS_INCLUDE_PATH and GEOS_LIBRARY_PATH environment variables (see below for notes on GEOS discovery)
- Run `pip install shapely --no-binary`
- Make sure the GEOS .dll files are available on the PATH

5.1.3 Installation for local development

This is similar to installing with a custom GEOS binary, but then instead of installing Shapely with pip from PyPI, you clone the package from Github:

```
$ git clone git@github.com:shapely/shapely.git
$ cd shapely/
```

Install it in development mode using `pip`:

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

For development, use of a virtual environment is strongly recommended. For example using `venv`:

```
$ python3 -m venv .
$ source bin/activate
(env) $ pip install -e .[test]
```

Or using `conda`:

```
$ conda create -n env python=3 geos numpy cython pytest
$ conda activate env
(env) $ pip install -e .
```

5.1.4 Testing Shapely

Shapely can be tested using `pytest`:

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

5.1.5 GEOS discovery (compile time)

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

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

Alternatively, you can specify where Shapely should look for GEOS library and header files using environment variables (on Linux/macOS):

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

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

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

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

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

5.1.6 GEOS discovery (runtime)

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

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

If you encounter exceptions like:

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

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

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

The filenames of the GEOS shared libraries are:

- On Linux: `libgeos-*.so.*, libgeos_c-*.so.*`
- On macOS: `libgeos.dylib, libgeos_c.dylib`

- On Windows: `geos-*.dll, geos_c-*.dll`

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

5.2 The Shapely User Manual

Author
Sean Gillies, <sean.gillies@gmail.com>

Version
2.0.3

Date
Apr 16, 2024

Abstract
This document explains how to use the Shapely Python package for computational geometry.

5.2.1 Introduction

Deterministic spatial analysis is an important component of computational approaches to problems in agriculture, ecology, epidemiology, sociology, and many other fields. What is the surveyed perimeter/area ratio of these patches of animal habitat? Which properties in this town intersect with the 50-year flood contour from this new flooding model? What are the extents of findspots for ancient ceramic wares with maker's marks "A" and "B", and where do the extents overlap? What's the path from home to office that best skirts identified zones of location based spam? These are just a few of the possible questions addressable using non-statistical spatial analysis, and more specifically, computational geometry.

Shapely is a Python package for set-theoretic analysis and manipulation of planar features using functions from the well known and widely deployed GEOS library. GEOS, a port of the Java Topology Suite (JTS), is the geometry engine of the PostGIS spatial extension for the PostgreSQL RDBMS. The designs of JTS and GEOS are largely guided by the Open Geospatial Consortium's Simple Features Access Specification[1] and Shapely adheres mainly to the same set of standard classes and operations. Shapely is thereby deeply rooted in the conventions of the geographic information systems (GIS) world, but aspires to be equally useful to programmers working on non-conventional problems.

The first premise of Shapely is that Python programmers should be able to perform PostGIS type geometry operations outside of an RDBMS. Not all geographic data originate or reside in a RDBMS or are best processed using SQL. We can load data into a spatial RDBMS to do work, but if there's no mandate to manage (the "M" in "RDBMS") the data over time in the database we're using the wrong tool for the job. The second premise is that the persistence, serialization, and map projection of features are significant, but orthogonal problems. You may not need a hundred GIS format readers and writers or the multitude of State Plane projections, and Shapely doesn't burden you with them. The third premise is that Python idioms trump GIS (or Java, in this case, since the GEOS library is derived from JTS, a Java project) idioms.

If you enjoy and profit from idiomatic Python, appreciate packages that do one thing well, and agree that a spatially enabled RDBMS is often enough the wrong tool for your computational geometry job, Shapely might be for you.

[1] John R. Herring, Ed., "OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture," Oct. 2006.

Spatial Data Model

The fundamental types of geometric objects implemented by Shapely are points, curves, and surfaces. Each is associated with three sets of (possibly infinite) points in the plane. The *interior*, *boundary*, and *exterior* sets of a feature are mutually exclusive and their union coincides with the entire plane[2].

- A *Point* has an *interior* set of exactly one point, a *boundary* set of exactly no points, and an *exterior* set of all other points. A *Point* has a topological dimension of 0.
- A *Curve* has an *interior* set consisting of the infinitely many points along its length (imagine a *Point* dragged in space), a *boundary* set consisting of its two end points, and an *exterior* set of all other points. A *Curve* has a topological dimension of 1.
- A *Surface* has an *interior* set consisting of the infinitely many points within (imagine a *Curve* dragged in space to cover an area), a *boundary* set consisting of one or more *Curves*, and an *exterior* set of all other points including those within holes that might exist in the surface. A *Surface* has a topological dimension of 2.

That may seem a bit esoteric, but will help clarify the meanings of Shapely's spatial predicates, and it's as deep into theory as this manual will go. Consequences of point-set theory, including some that manifest themselves as "gotchas", for different classes will be discussed later in this manual.

The point type is implemented by a *Point* class; curve by the *LineString* and *LinearRing* classes; and surface by a *Polygon* class. Shapely implements no smooth (*i.e.* having continuous tangents) curves. All curves must be approximated by linear splines. All rounded patches must be approximated by regions bounded by linear splines.

Collections of points are implemented by a *MultiPoint* class, collections of curves by a *MultiLineString* class, and collections of surfaces by a *MultiPolygon* class. These collections aren't computationally significant, but are useful for modeling certain kinds of features. A Y-shaped line feature, for example, is well modeled as a whole by a *MultiLineString*.

The standard data model has additional constraints specific to certain types of geometric objects that will be discussed in following sections of this manual.

See also https://web.archive.org/web/20160719195511/http://www.vividsolutions.com/jts/discussion.htm for more illustrations of this data model.

Relationships

The spatial data model is accompanied by a group of natural language relationships between geometric objects – *contains*, *intersects*, *overlaps*, *touches*, etc. – and a theoretical framework for understanding them using the 3x3 matrix of the mutual intersections of their component point sets[3]: the DE-9IM. A comprehensive review of the relationships in terms of the DE-9IM is found in[4] and will not be reiterated in this manual.

[2] M.J. Egenhofer and John R. Herring, Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases, Orono, ME: University of Maine, 1991.

[3] E. Clementini, P. Di Felice, and P. van Oosterom, "A Small Set of Formal Topological Relationships Suitable for End-User Interaction," Third International Symposium on Large Spatial Databases (SSD). Lecture Notes in Computer Science no. 692, David Abel and Beng Chin Ooi, Eds., Singapore: Springer Verlag, 1993, pp. 277-295.

[4] C. Strobl, "Dimensionally Extended Nine-Intersection Model (DE-9IM)," Encyclopedia of GIS, S. Shekhar and H. Xiong, Eds., Springer, 2008, pp. 240-245. [PDF]

Operations

Following the JTS technical specs[5], this manual will make a distinction between constructive (*buffer*, *convex hull*) and set-theoretic operations (*intersection*, *union*, etc.). The individual operations will be fully described in a following section of the manual.

Coordinate Systems

Even though the Earth is not flat – and for that matter not exactly spherical – there are many analytic problems that can be approached by transforming Earth features to a Cartesian plane, applying tried and true algorithms, and then transforming the results back to geographic coordinates. This practice is as old as the tradition of accurate paper maps.

Shapely does not support coordinate system transformations. All operations on two or more features presume that the features exist in the same Cartesian plane.

5.2.2 Geometric Objects

Geometric objects are created in the typical Python fashion, using the classes themselves as instance factories. A few of their intrinsic properties will be discussed in this sections, others in the following sections on operations and serializations.

Instances of `Point`, `LineString`, and `LinearRing` have as their most important attribute a finite sequence of coordinates that determines their interior, boundary, and exterior point sets. A line string can be determined by as few as 2 points, but contains an infinite number of points. Coordinate sequences are immutable. A third z coordinate value may be used when constructing instances, but has no effect on geometric analysis. All operations are performed in the x-y plane.

In all constructors, numeric values are converted to type `float`. In other words, `Point(0, 0)` and `Point(0.0, 0.0)` produce geometrically equivalent instances. Shapely does not check the topological simplicity or validity of instances when they are constructed as the cost is unwarranted in most cases. Validating factories are easily implemented using the :attr:`is_valid` predicate by users that require them.

Note: Shapely is a planar geometry library and z, the height above or below the plane, is ignored in geometric analysis. There is a potential pitfall for users here: coordinate tuples that differ only in z are not distinguished from each other and their application can result in surprisingly invalid geometry objects. For example, `LineString([(0, 0, 0), (0, 0, 1)])` does not return a vertical line of unit length, but an invalid line in the plane with zero length. Similarly, `Polygon([(0, 0, 0), (0, 0, 1), (1, 1, 1)])` is not bounded by a closed ring and is invalid.

General Attributes and Methods

`object.`**`area`**

Returns the area (`float`) of the object.

`object.`**`bounds`**

Returns a `(minx, miny, maxx, maxy)` tuple (`float` values) that bounds the object.

`object.`**`length`**

Returns the length (`float`) of the object.

[5] Martin Davis, "JTS Technical Specifications," Mar. 2003. [PDF]

object.**minimum_clearance**

Returns the smallest distance by which a node could be moved to produce an invalid geometry.

This can be thought of as a measure of the robustness of a geometry, where larger values of minimum clearance indicate a more robust geometry. If no minimum clearance exists for a geometry, such as a point, this will return *math.infinity*.

New in Shapely 1.7.1

Requires GEOS 3.6 or higher.

```
>>> from shapely import Polygon
>>> Polygon([[0, 0], [1, 0], [1, 1], [0, 1], [0, 0]]).minimum_clearance
1.0
```

object.**geom_type**

Returns a string specifying the *Geometry Type* of the object in accordance with[Page 20, 1].

```
>>> from shapely import Point, LineString
>>> Point(0, 0).geom_type
'Point'
```

object.**distance**(*other*)

Returns the minimum distance (`float`) to the *other* geometric object.

```
>>> Point(0,0).distance(Point(1,1))
1.4142135623730951
```

object.**hausdorff_distance**(*other*)

Returns the Hausdorff distance (`float`) to the *other* geometric object. The Hausdorff distance between two geometries is the furthest distance that a point on either geometry can be from the nearest point to it on the other geometry.

New in Shapely 1.6.0

```
>>> point = Point(1, 1)
>>> line = LineString([(2, 0), (2, 4), (3, 4)])
>>> point.hausdorff_distance(line)
3.605551275463989
>>> point.distance(Point(3, 4))
3.605551275463989
```

object.**representative_point**()

Returns a cheaply computed point that is guaranteed to be within the geometric object.

Note: This is not in general the same as the centroid.

```
>>> donut = Point(0, 0).buffer(2.0).difference(Point(0, 0).buffer(1.0))
>>> donut.centroid
<POINT (0 0)>
>>> donut.representative_point()
<POINT (1.498 0.049)>
```

Points

class Point(*coordinates*)

The *Point* constructor takes positional coordinate values or point tuple parameters.

```
>>> from shapely import Point
>>> point = Point(0.0, 0.0)
>>> q = Point((0.0, 0.0))
```

A *Point* has zero area and zero length.

```
>>> point.area
0.0
>>> point.length
0.0
```

Its *x*-*y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> point.bounds
(0.0, 0.0, 0.0, 0.0)
```

Coordinate values are accessed via *coords*, *x*, *y*, and *z* properties.

```
>>> list(point.coords)
[(0.0, 0.0)]
>>> point.x
0.0
>>> point.y
0.0
```

Coordinates may also be sliced. *New in version 1.2.14*.

```
>>> point.coords[:]
[(0.0, 0.0)]
```

The *Point* constructor also accepts another *Point* instance, thereby making a copy.

```
>>> Point(point)
<POINT (0 0)>
```

LineStrings

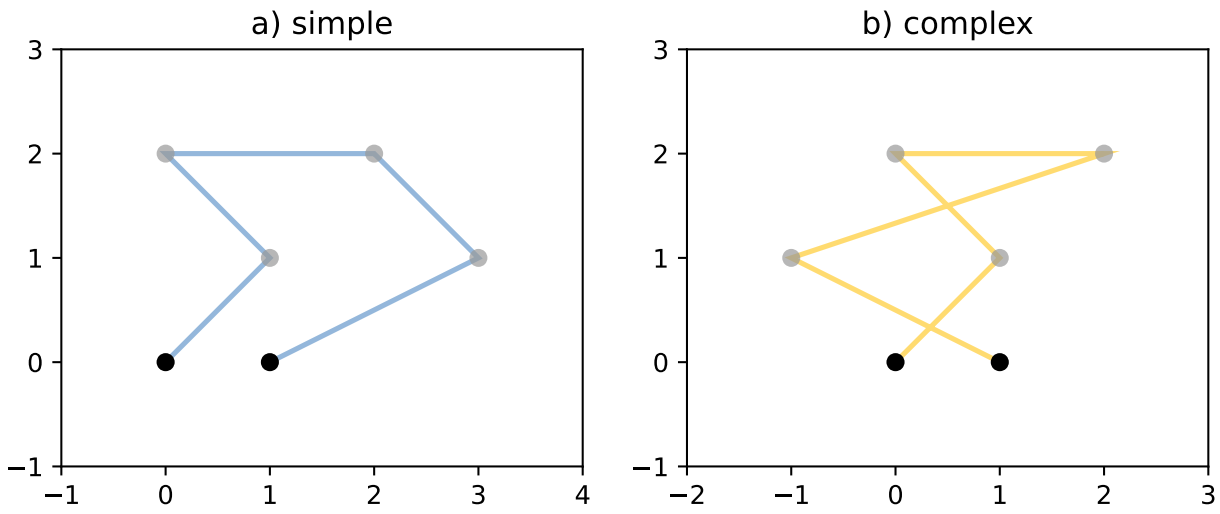
class LineString(*coordinates*)

The *LineString* constructor takes an ordered sequence of 2 or more `(x, y[, z])` point tuples.

The constructed *LineString* object represents one or more connected linear splines between the points. Repeated points in the ordered sequence are allowed, but may incur performance penalties and should be avoided. A *LineString* may cross itself (*i.e.* be *complex* and not *simple*).

Figure 1. A simple *LineString* on the left, a complex *LineString* on the right. The (*MultiPoint*) boundary of each is shown in black, the other points that describe the lines are shown in grey.

A *LineString* has zero area and non-zero length.




```
>>> from shapely import LineString
>>> line = LineString([(0, 0), (1, 1)])
>>> line.area
0.0
>>> line.length
1.4142135623730951
```

Its *x-y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> line.bounds
(0.0, 0.0, 1.0, 1.0)
```

The defining coordinate values are accessed via the *coords* property.

```
>>> len(line.coords)
2
>>> list(line.coords)
[(0.0, 0.0), (1.0, 1.0)]
```

Coordinates may also be sliced. *New in version 1.2.14*.

```
>>> line.coords[:]
[(0.0, 0.0), (1.0, 1.0)]
>>> line.coords[1:]
[(1.0, 1.0)]
```

The constructor also accepts another *LineString* instance, thereby making a copy.

```
>>> LineString(line)
<LINESTRING (0 0, 1 1)>
```

A *LineString* may also be constructed using a sequence of mixed *Point* instances or coordinate tuples. The individual coordinates are copied into the new object.

```
>>> LineString([Point(0.0, 1.0), (2.0, 3.0), Point(4.0, 5.0)])
<LINESTRING (0 1, 2 3, 4 5)>
```

LinearRings

class LinearRing(*coordinates*)

The *LinearRing* constructor takes an ordered sequence of `(x, y[, z])` point tuples.

The sequence may be explicitly closed by passing identical values in the first and last indices. Otherwise, the sequence will be implicitly closed by copying the first tuple to the last index. As with a *LineString*, repeated points in the ordered sequence are allowed, but may incur performance penalties and should be avoided. A *LinearRing* may not cross itself, and may not touch itself at a single point.

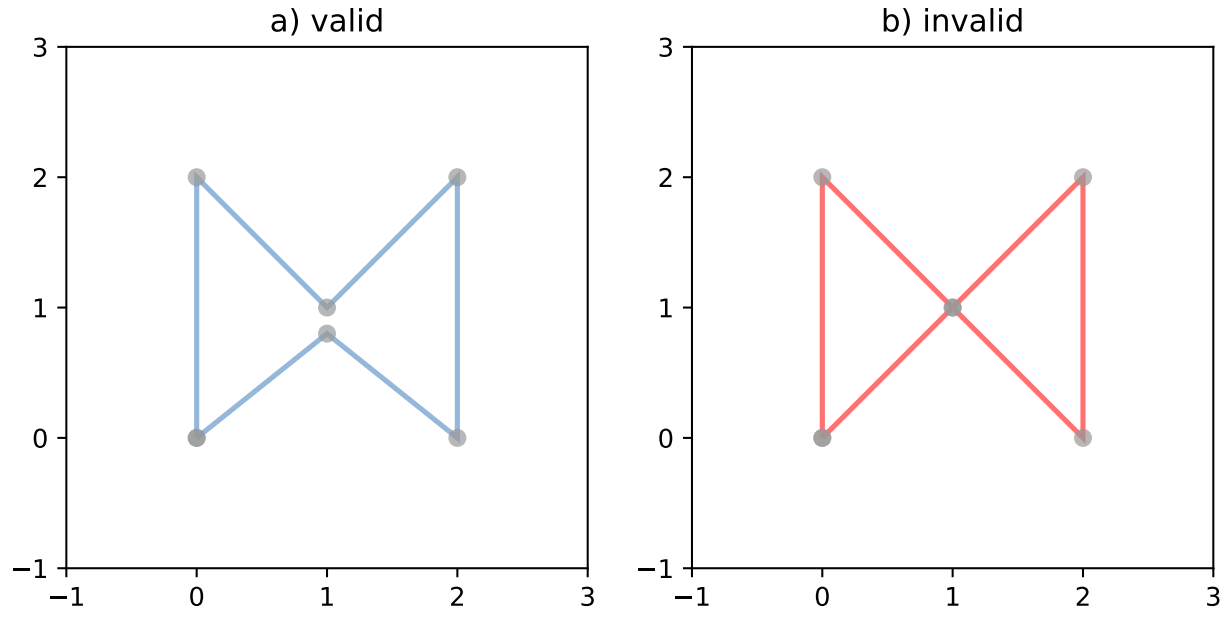


Figure 2. A valid *LinearRing* on the left, an invalid self-touching *LinearRing* on the right. The points that describe the rings are shown in grey. A ring's boundary is *empty*.

Note: Shapely will not prevent the creation of such rings, but exceptions will be raised when they are operated on.

A *LinearRing* has zero area and non-zero length.

```
>>> from shapely import LinearRing
>>> ring = LinearRing([(0, 0), (1, 1), (1, 0)])
>>> ring.area
0.0
>>> ring.length
3.414213562373095
```

Its *x*-*y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> ring.bounds
(0.0, 0.0, 1.0, 1.0)
```

Defining coordinate values are accessed via the *coords* property.

```
>>> len(ring.coords)
4
>>> list(ring.coords)
[(0.0, 0.0), (1.0, 1.0), (1.0, 0.0), (0.0, 0.0)]
```

The *LinearRing* constructor also accepts another *LineString* or *LinearRing* instance, thereby making a copy.

```
>>> LinearRing(ring)
<LINEARRING (0 0, 1 1, 1 0, 0 0)>
```

As with *LineString*, a sequence of *Point* instances is not a valid constructor parameter.

Polygons

class Polygon(*shell*[, *holes=None*])

The *Polygon* constructor takes two positional parameters. The first is an ordered sequence of `(x, y[, z])` point tuples and is treated exactly as in the *LinearRing* case. The second is an optional unordered sequence of ring-like sequences specifying the interior boundaries or "holes" of the feature.

Rings of a *valid Polygon* may not cross each other, but may touch at a single point only. Again, Shapely will not prevent the creation of invalid features, but exceptions will be raised when they are operated on.

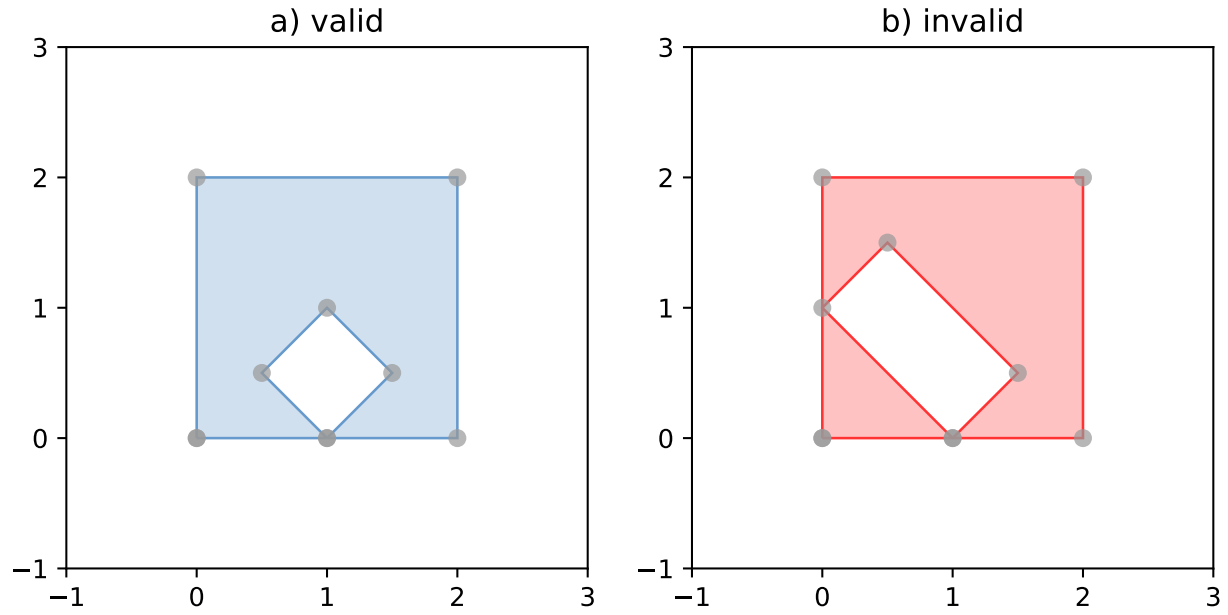
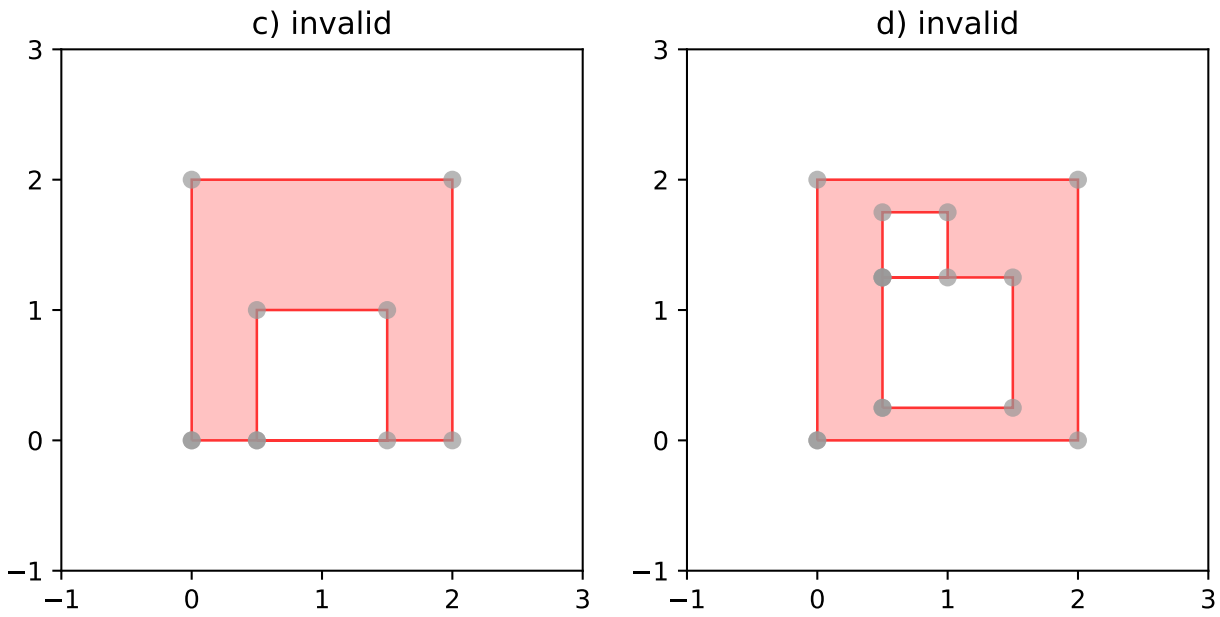


Figure 3. On the left, a valid *Polygon* with one interior ring that touches the exterior ring at one point, and on the right a *Polygon* that is *invalid* because its interior ring touches the exterior ring at more than one point. The points that describe the rings are shown in grey.

Figure 4. On the left, a *Polygon* that is *invalid* because its exterior and interior rings touch along a line, and on the right, a *Polygon* that is *invalid* because its interior rings touch along a line.

A *Polygon* has non-zero area and non-zero length.




```
>>> from shapely import Polygon
>>> polygon = Polygon([(0, 0), (1, 1), (1, 0)])
>>> polygon.area
0.5
>>> polygon.length
3.414213562373095
```

Its *x-y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> polygon.bounds
(0.0, 0.0, 1.0, 1.0)
```

Component rings are accessed via *exterior* and *interiors* properties.

```
>>> list(polygon.exterior.coords)
[(0.0, 0.0), (1.0, 1.0), (1.0, 0.0), (0.0, 0.0)]
>>> list(polygon.interiors)
[]
```

The *Polygon* constructor also accepts instances of *LineString* and *LinearRing*.

```
>>> coords = [(0, 0), (1, 1), (1, 0)]
>>> r = LinearRing(coords)
>>> s = Polygon(r)
>>> s.area
0.5
>>> t = Polygon(s.buffer(1.0).exterior, [r])
>>> t.area
6.550762052919032 5
```

Rectangular polygons occur commonly, and can be conveniently constructed using the *shapely.geometry.box()* function.

shapely.geometry.**box**(*minx*, *miny*, *maxx*, *maxy*, *ccw=True*)

Makes a rectangular polygon from the provided bounding box values, with counter-clockwise order by default.

New in version 1.2.9.

For example:

```
>>> from shapely import box
>>> b = box(0.0, 0.0, 1.0, 1.0)
>>> b
<POLYGON ((1 0, 1 1, 0 1, 0 0, 1 0))>
>>> list(b.exterior.coords)
[(1.0, 0.0), (1.0, 1.0), (0.0, 1.0), (0.0, 0.0), (1.0, 0.0)]
```

This is the first appearance of an explicit polygon handedness in Shapely.

To obtain a polygon with a known orientation, use *shapely.geometry.polygon.orient()*:

shapely.geometry.polygon.**orient**(*polygon*, *sign=1.0*)

Returns a properly oriented copy of the given polygon. The signed area of the result will have the given sign. A sign of 1.0 means that the coordinates of the product's exterior ring will be oriented counter-clockwise and the interior rings (holes) will be oriented clockwise.

New in version 1.2.10.

Collections

Heterogeneous collections of geometric objects may result from some Shapely operations. For example, two *LineStrings* may intersect along a line and at a point. To represent these kind of results, Shapely provides frozenset-like, immutable collections of geometric objects. The collections may be homogeneous (*MultiPoint* etc.) or heterogeneous.

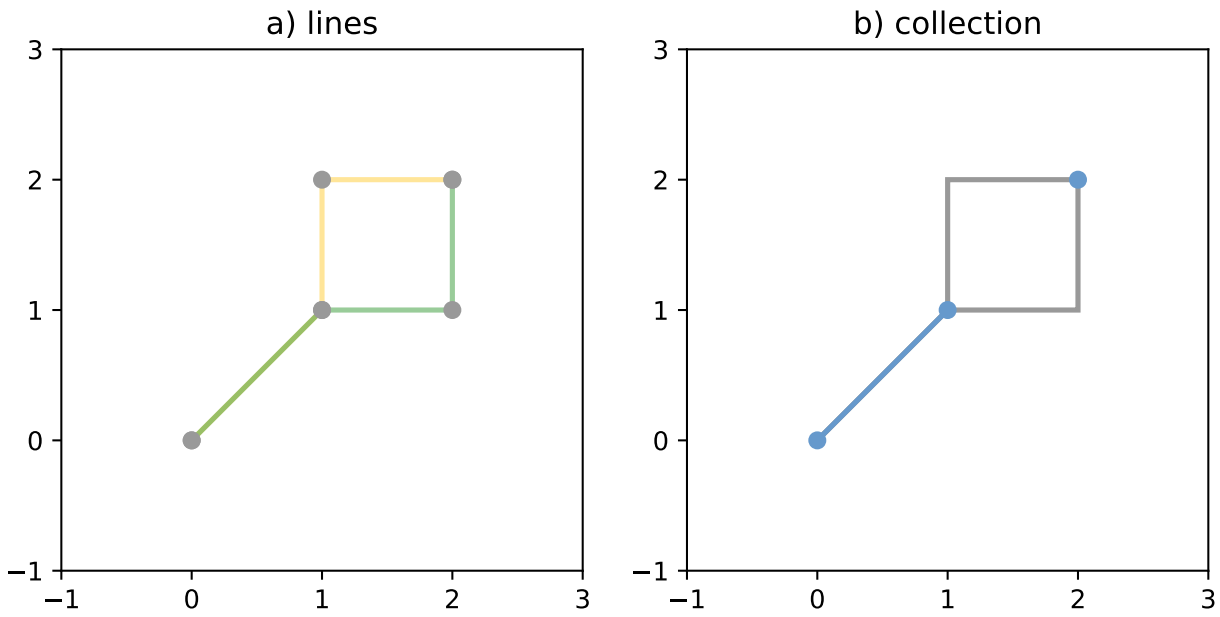
```
>>> a = LineString([(0, 0), (1, 1), (1,2), (2,2)])
>>> b = LineString([(0, 0), (1, 1), (2,1), (2,2)])
>>> x = a.intersection(b)
>>> x
<GEOMETRYCOLLECTION (LINESTRING (0 0, 1 1), POINT (2 2))>
>>> list(x.geoms)
[<LINESTRING (0 0, 1 1)>, <POINT (2 2)>]
```

Figure 5. a) a green and a yellow line that intersect along a line and at a single point; b) the intersection (in blue) is a collection containing one *LineString* and one *Point*.

Members of a *GeometryCollection* are accessed via the `geoms` property.

```
>>> list(x.geoms)
[<LINESTRING (0 0, 1 1)>, <POINT (2 2)>]
```

Note: When possible, it is better to use one of the homogeneous collection types described below.



Collections of Points

class MultiPoint(*points*)

The *MultiPoint* constructor takes a sequence of `(x, y[, z ])` point tuples.

A *MultiPoint* has zero area and zero length.

```
>>> from shapely import MultiPoint
>>> points = MultiPoint([(0.0, 0.0), (1.0, 1.0)])
>>> points.area
0.0
>>> points.length
0.0
```

Its *x-y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> points.bounds
(0.0, 0.0, 1.0, 1.0)
```

Members of a multi-point collection are accessed via the `geoms` property.

```
>>> list(points.geoms)
[<POINT (0 0)>, <POINT (1 1)>]
```

The constructor also accepts another *MultiPoint* instance or an unordered sequence of *Point* instances, thereby making copies.

```
>>> MultiPoint([Point(0, 0), Point(1, 1)])
<MULTIPOINT (0 0, 1 1)>
```

Collections of Lines

class MultiLineString(*lines*)

The *MultiLineString* constructor takes a sequence of line-like sequences or objects.

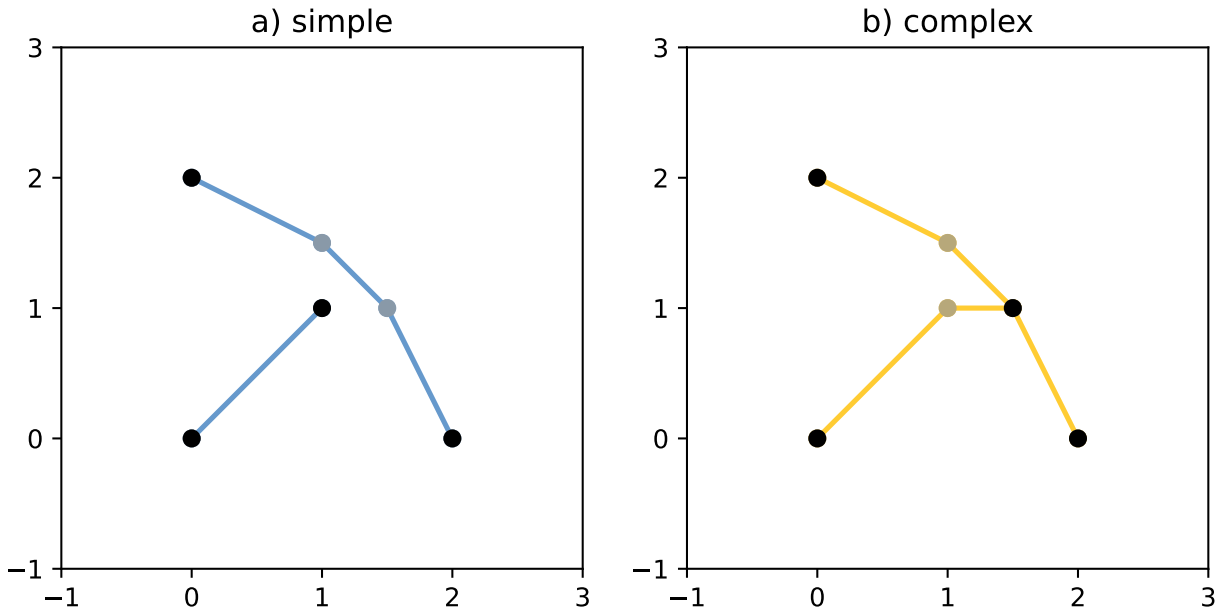


Figure 6. On the left, a *simple*, disconnected *MultiLineString*, and on the right, a non-simple *MultiLineString*. The points defining the objects are shown in gray, the boundaries of the objects in black.

A *MultiLineString* has zero area and non-zero length.

```
>>> from shapely import MultiLineString
>>> coords = [((0, 0), (1, 1)), ((-1, 0), (1, 0))]
>>> lines = MultiLineString(coords)
>>> lines.area
0.0
>>> lines.length
3.414213562373095
```

Its *x*-*y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> lines.bounds
(-1.0, 0.0, 1.0, 1.0)
```

Its members are instances of *LineString* and are accessed via the `geoms` property.

```
>>> len(lines.geoms)
2
>>> print(list(lines.geoms))
[<LINESTRING (0 0, 1 1)>, <LINESTRING (-1 0, 1 0)>]
```

The constructor also accepts another instance of *MultiLineString* or an unordered sequence of *LineString* instances, thereby making copies.

```
>>> MultiLineString(lines)
<MULTILINESTRING ((0 0, 1 1), (-1 0, 1 0))>
>>> MultiLineString(lines.geoms)
<MULTILINESTRING ((0 0, 1 1), (-1 0, 1 0))>
```

Collections of Polygons

class MultiPolygon(*polygons*)

The *MultiPolygon* constructor takes a sequence of exterior ring and hole list tuples: [((a1, ..., aM), [(b1, ..., bN), ...]), ...].

More clearly, the constructor also accepts an unordered sequence of *Polygon* instances, thereby making copies.

```
>>> from shapely import MultiPolygon
>>> polygons = MultiPolygon([polygon, s, t])
>>> len(polygons.geoms)
3
```

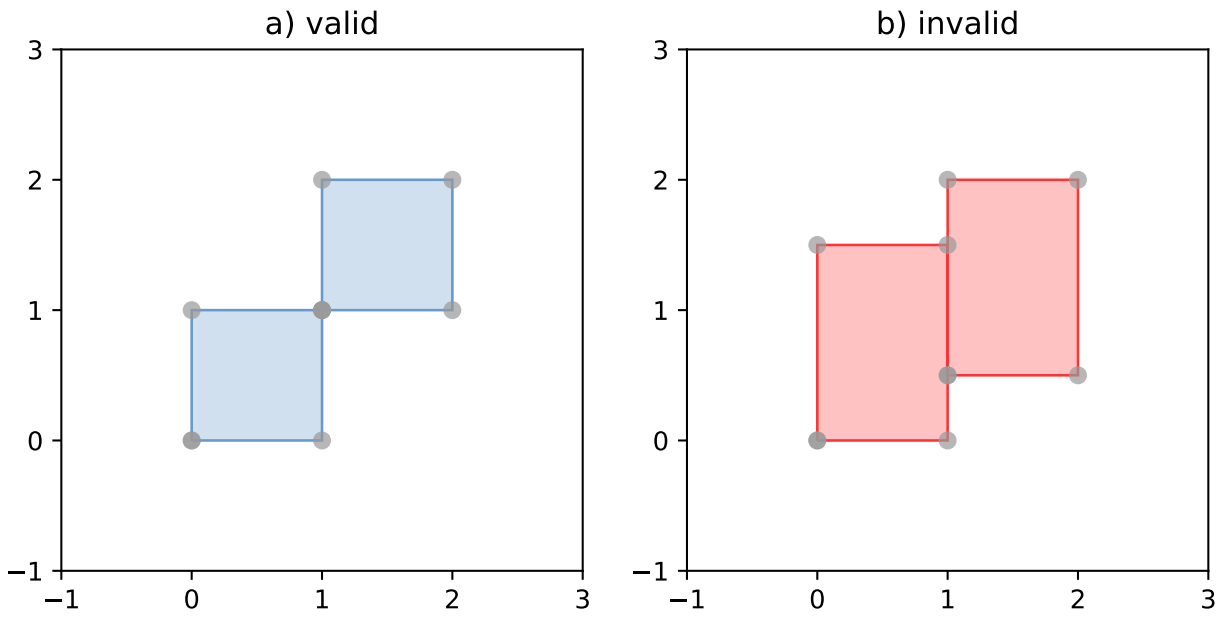


Figure 7. On the left, a *valid MultiPolygon* with 2 members, and on the right, a *MultiPolygon* that is invalid because its members touch at an infinite number of points (along a line).

Its *x*-*y* bounding box is a `(minx, miny, maxx, maxy)` tuple.

```
>>> polygons.bounds
(-1.0, -1.0, 2.0, 2.0)
```

Its members are instances of *Polygon* and are accessed via the `geoms` property.

```
>>> len(polygons.geoms)
3
```

Empty features

An "empty" feature is one with a point set that coincides with the empty set; not `None`, but like `set([])`. Empty features can be created by calling the various constructors with no arguments. Almost no operations are supported by empty features.

```
>>> line = LineString()
>>> line.is_empty
True
>>> line.length
0.0
>>> line.bounds
(nan, nan, nan, nan)
>>> list(line.coords)
[]
```

Coordinate sequences

The list of coordinates that describe a geometry are represented as the `CoordinateSequence` object. These sequences should not be initialised directly, but can be accessed from an existing geometry as the `Geometry.coords` property.

```
>>> line = LineString([(0, 1), (2, 3), (4, 5)])
>>> line.coords
<shapely.coords.CoordinateSequence object at ...>
```

Coordinate sequences can be indexed, sliced and iterated over as if they were a list of coordinate tuples.

```
>>> line.coords[0]
(0.0, 1.0)
>>> line.coords[1:]
[(2.0, 3.0), (4.0, 5.0)]
>>> for x, y in line.coords:
...     print("x={}, y={}".format(x, y))
...
x=0.0, y=1.0
x=2.0, y=3.0
x=4.0, y=5.0
```

Polygons have a coordinate sequence for their exterior and each of their interior rings.

```
>>> poly = Polygon([(0, 0), (0, 1), (1, 1), (0, 0)])
>>> poly.exterior.coords
<shapely.coords.CoordinateSequence object at ...>
```

Multipart geometries do not have a coordinate sequence. Instead the coordinate sequences are stored on their component geometries.

```
>>> p = MultiPoint([(0, 0), (1, 1), (2, 2)])
>>> p.geoms[2].coords
<shapely.coords.CoordinateSequence object at ...>
```

Linear Referencing Methods

It can be useful to specify position along linear features such as *LineStrings* and *MultiLineStrings* with a 1-dimensional referencing system. Shapely supports linear referencing based on length or distance, evaluating the distance along a geometric object to the projection of a given point, or the point at a given distance along the object.

`object.`**`interpolate`**(*distance*[, *normalized=False*])

Return a point at the specified distance along a linear geometric object.

If the *normalized* arg is `True`, the distance will be interpreted as a fraction of the geometric object's length.

```
>>> ip = LineString([(0, 0), (0, 1), (1, 1)]).interpolate(1.5)
>>> ip
<POINT (0.5 1)>
>>> LineString([(0, 0), (0, 1), (1, 1)]).interpolate(0.75, normalized=True)
<POINT (0.5 1)>
```

`object.`**`project`**(*other*[, *normalized=False*])

Returns the distance along this geometric object to a point nearest the *other* object.

If the *normalized* arg is `True`, return the distance normalized to the length of the object. The *project()* method is the inverse of *interpolate()*.

```
>>> LineString([(0, 0), (0, 1), (1, 1)]).project(ip)
1.5
>>> LineString([(0, 0), (0, 1), (1, 1)]).project(ip, normalized=True)
0.75
```

For example, the linear referencing methods might be used to cut lines at a specified distance.

```
def cut(line, distance):
    # Cuts a line in two at a distance from its starting point
    if distance <= 0.0 or distance >= line.length:
        return [LineString(line)]
    coords = list(line.coords)
    for i, p in enumerate(coords):
        pd = line.project(Point(p))
        if pd == distance:
            return [
                LineString(coords[:i+1]),
                LineString(coords[i:])]
        if pd > distance:
            cp = line.interpolate(distance)
            return [
                LineString(coords[:i] + [(cp.x, cp.y)]),
                LineString([(cp.x, cp.y)] + coords[i:])]
```

```
>>> line = LineString([(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0)])
>>> print([list(x.coords) for x in cut(line, 1.0)])
[[(0.0, 0.0), (1.0, 0.0)],
 [(1.0, 0.0), (2.0, 0.0), (3.0, 0.0), (4.0, 0.0), (5.0, 0.0)]]
>>> print([list(x.coords) for x in cut(line, 2.5)])
[[(0.0, 0.0), (1.0, 0.0), (2.0, 0.0), (2.5, 0.0)],
 [(2.5, 0.0), (3.0, 0.0), (4.0, 0.0), (5.0, 0.0)]]
```

5.2.3 Predicates and Relationships

Objects of the types explained in *Geometric Objects* provide standard[Page 20, 1] predicates as attributes (for unary predicates) and methods (for binary predicates). Whether unary or binary, all return `True` or `False`.

Unary Predicates

Standard unary predicates are implemented as read-only property attributes. An example will be shown for each.

`object.`**`has_z`**

Returns `True` if the feature has not only *x* and *y*, but also *z* coordinates for 3D (or so-called, 2.5D) geometries.

```
>>> Point(0, 0).has_z
False
>>> Point(0, 0, 0).has_z
True
```

`object.`**`is_ccw`**

Returns `True` if coordinates are in counter-clockwise order (bounding a region with positive signed area). This method applies to *LinearRing* objects only.

New in version 1.2.10.

```
>>> LinearRing([(1,0), (1,1), (0,0)]).is_ccw
True
```

A ring with an undesired orientation can be reversed like this:

```
>>> ring = LinearRing([(0,0), (1,1), (1,0)])
>>> ring.is_ccw
False
>>> ring2 = LinearRing(list(ring.coords)[::-1])
>>> ring2.is_ccw
True
```

`object.`**`is_empty`**

Returns `True` if the feature's *interior* and *boundary* (in point set terms) coincide with the empty set.

```
>>> Point().is_empty
True
>>> Point(0, 0).is_empty
False
```

Note: With the help of the `operator` module's `attrgetter()` function, unary predicates such as `is_empty` can be easily used as predicates for the built in `filter()`.

```
>>> from operator import attrgetter
>>> empties = filter(attrgetter('is_empty'), [Point(), Point(0, 0)])
>>> len(list(empties))
1
```

`object.is_ring`

Returns `True` if the feature is a closed and simple `LineString`. A closed feature's *boundary* coincides with the empty set.

```
>>> LineString([(0, 0), (1, 1), (1, -1)]).is_ring
False
>>> LinearRing([(0, 0), (1, 1), (1, -1)]).is_ring
True
```

This property is applicable to *LineString* and *LinearRing* instances, but meaningless for others.

`object.is_simple`

Returns `True` if the feature does not cross itself.

Note: The simplicity test is meaningful only for *LineStrings* and *LinearRings*.

```
>>> LineString([(0, 0), (1, 1), (1, -1), (0, 1)]).is_simple
False
```

Operations on non-simple *LineStrings* are fully supported by Shapely.

`object.is_valid`

Returns `True` if a feature is "valid" in the sense of[Page 20, 1].

Note: The validity test is meaningful only for *Polygons* and *MultiPolygons*. `True` is always returned for other types of geometries.

A valid *Polygon* may not possess any overlapping exterior or interior rings. A valid *MultiPolygon* may not collect any overlapping polygons. Operations on invalid features may fail.

```
>>> MultiPolygon([Point(0, 0).buffer(2.0), Point(1, 1).buffer(2.0)]).is_valid
False
```

The two points above are close enough that the polygons resulting from the buffer operations (explained in a following section) overlap.

Note: The `is_valid` predicate can be used to write a validating decorator that could ensure that only valid objects are returned from a constructor function.

```
from functools import wraps
def validate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        ob = func(*args, **kwargs)
        if not ob.is_valid:
            raise TopologicalError(
                "Given arguments do not determine a valid geometric object")
        return ob
    return wrapper
```

```
>>> @validate
... def ring(coordinates):
...     return LinearRing(coordinates)
...
>>> coords = [(0, 0), (1, 1), (1, -1), (0, 1)]
>>> ring(coords)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in wrapper
shapely.geos.TopologicalError: Given arguments do not determine a valid geometric object
```

Binary Predicates

Standard binary predicates are implemented as methods. These predicates evaluate topological, set-theoretic relationships. In a few cases the results may not be what one might expect starting from different assumptions. All take another geometric object as argument and return `True` or `False`.

`object.__eq__`(*other*)

Returns `True` if the two objects are of the same geometric type, and the coordinates of the two objects match precisely.

`object.equals`(*other*)

Returns `True` if the set-theoretic *boundary*, *interior*, and *exterior* of the object coincide with those of the other.

The coordinates passed to the object constructors are of these sets, and determine them, but are not the entirety of the sets. This is a potential "gotcha" for new users. Equivalent lines, for example, can be constructed differently.

```
>>> a = LineString([(0, 0), (1, 1)])
>>> b = LineString([(0, 0), (0.5, 0.5), (1, 1)])
>>> c = LineString([(0, 0), (0, 0), (1, 1)])
>>> a.equals(b)
True
>>> a == b
False
>>> b.equals(c)
True
>>> b == c
False
```

`object.equals_exact`(*other*, *tolerance*)

Returns `True` if the object is within a specified *tolerance*.

`object.contains`(*other*)

Returns `True` if no points of *other* lie in the exterior of the *object* and at least one point of the interior of *other* lies in the interior of *object*.

This predicate applies to all types, and is inverse to *within()*. The expression `a.contains(b) == b.within(a)` always evaluates to `True`.

```
>>> coords = [(0, 0), (1, 1)]
>>> LineString(coords).contains(Point(0.5, 0.5))
True
>>> Point(0.5, 0.5).within(LineString(coords))
True
```

A line's endpoints are part of its *boundary* and are therefore not contained.

```
>>> LineString(coords).contains(Point(1.0, 1.0))
False
```

Note: Binary predicates can be used directly as predicates for `filter()` or `itertools.ifilter()`.

```
>>> line = LineString(coords)
>>> contained = list(filter(line.contains, [Point(), Point(0.5, 0.5)]))
>>> len(contained)
1
>>> contained
[<POINT (0.5 0.5)>]
```

`object.`**`covers`**`(`*other*`)`

Returns `True` if every point of *other* is a point on the interior or boundary of *object*. This is similar to `object.contains(other)` except that this does not require any interior points of *other* to lie in the interior of *object*.

`object.`**`covered_by`**`(`*other*`)`

Returns `True` if every point of *object* is a point on the interior or boundary of *other*. This is equivalent to `other.covers(object)`.

New in version 1.8.

`object.`**`crosses`**`(`*other*`)`

Returns `True` if the *interior* of the object intersects the *interior* of the other but does not contain it, and the dimension of the intersection is less than the dimension of the one or the other.

```
>>> LineString(coords).crosses(LineString([(0, 1), (1, 0)]))
True
```

A line does not cross a point that it contains.

```
>>> LineString(coords).crosses(Point(0.5, 0.5))
False
```

`object.`**`disjoint`**`(`*other*`)`

Returns `True` if the *boundary* and *interior* of the object do not intersect at all with those of the other.

```
>>> Point(0, 0).disjoint(Point(1, 1))
True
```

This predicate applies to all types and is the inverse of *intersects()*.

`object.`**`intersects`**`(`*other*`)`

Returns `True` if the *boundary* or *interior* of the object intersect in any way with those of the other.

In other words, geometric objects intersect if they have any boundary or interior point in common.

`object.`**`overlaps`**`(`*other*`)`

Returns `True` if the geometries have more than one but not all points in common, have the same dimension, and the intersection of the interiors of the geometries has the same dimension as the geometries themselves.

`object.touches(other)`

Returns `True` if the objects have at least one point in common and their interiors do not intersect with any part of the other.

Overlapping features do not therefore *touch*, another potential "gotcha". For example, the following lines touch at `(1, 1)`, but do not overlap.

```
>>> a = LineString([(0, 0), (1, 1)])
>>> b = LineString([(1, 1), (2, 2)])
>>> a.touches(b)
True
```

`object.within(other)`

Returns `True` if the object's *boundary* and *interior* intersect only with the *interior* of the other (not its *boundary* or *exterior*).

This applies to all types and is the inverse of *contains()*.

Used in a `sorted()` *key*, *within()* makes it easy to spatially sort objects. Let's say we have 4 stereotypic features: a point that is contained by a polygon which is itself contained by another polygon, and a free spirited point contained by none

```
>>> a = Point(2, 2)
>>> b = Polygon([[1, 1], [1, 3], [3, 3], [3, 1]])
>>> c = Polygon([[0, 0], [0, 4], [4, 4], [4, 0]])
>>> d = Point(-1, -1)
```

and that copies of these are collected into a list

```
>>> features = [c, a, d, b, c]
```

that we'd prefer to have ordered as `[d, c, c, b, a]` in reverse containment order. As explained in the Python Sorting HowTo, we can define a key function that operates on each list element and returns a value for comparison. Our key function will be a wrapper class that implements `__lt__()` using Shapely's binary *within()* predicate.

```
>>> class Within:
...     def __init__(self, o):
...         self.o = o
...     def __lt__(self, other):
...         return self.o.within(other.o)
```

As the howto says, the *less than* comparison is guaranteed to be used in sorting. That's what we'll rely on to spatially sort. Trying it out on features *d* and *c*, we see that it works.

```
>>> Within(d) < Within(c)
False
```

It also works on the list of features, producing the order we want.

```
>>> [d, c, c, b, a] == sorted(features, key=Within, reverse=True)
True
```

DE-9IM Relationships

The *relate()* method tests all the DE-9IM[Page 21, 4] relationships between objects, of which the named relationship predicates above are a subset.

`object.`**`relate`**`(`*other*`)`

Returns a string representation of the DE-9IM matrix of relationships between an object's *interior*, *boundary*, *exterior* and those of another geometric object.

The named relationship predicates (*contains()*, etc.) are typically implemented as wrappers around *relate()*.

Two different points have mainly `F` (false) values in their matrix; the intersection of their *external* sets (the 9th element) is a `2` dimensional object (the rest of the plane). The intersection of the *interior* of one with the *exterior* of the other is a `0` dimensional object (3rd and 7th elements of the matrix).

```
>>> Point(0, 0).relate(Point(1, 1))
'FF0FFF0F2'
```

The matrix for a line and a point on the line has more "true" (not `F`) elements.

```
>>> Point(0, 0).relate(LineString([(0, 0), (1, 1)]))
'F0FFFF102'
```

`object.`**`relate_pattern`**`(`*other*, *pattern*`)`

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

The *relate_pattern()* 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 (`T` or `F`), or a wildcard (`*`). For example, the pattern for the *within* predicate is `T*****FF*`.

```
>>> point = Point(0.5, 0.5)
>>> square = Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])
>>> square.relate_pattern(point, 'T*****FF*')
True
>>> point.within(square)
True
```

Note that the order or the geometries is significant, as demonstrated below. In this example the square contains the point, but the point does not contain the square.

```
>>> point.relate(square)
'0FFFFF212'
>>> square.relate(point)
'0F2FF1FF2'
```

Further discussion of the DE-9IM matrix is beyond the scope of this manual. See[Page 21, 4] and https://pypi.org/project/de9im/.

5.2.4 Spatial Analysis Methods

As well as boolean attributes and methods, Shapely provides analysis methods that return new geometric objects.

Set-theoretic Methods

Almost every binary predicate method has a counterpart that returns a new geometric object. In addition, the set-theoretic *boundary* of an object is available as a read-only attribute.

Note: These methods will *always* return a geometric object. An intersection of disjoint geometries for example will return an empty *GeometryCollection*, not *None* or *False*. To test for a non-empty result, use the geometry's *is_empty* property.

`object.`**boundary**

Returns a lower dimensional object representing the object's set-theoretic *boundary*.

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty collection.

```
>>> coords = [((0, 0), (1, 1)), ((-1, 0), (1, 0))]
>>> lines = MultiLineString(coords)
>>> lines.boundary
<MULTIPOINT (-1 0, 0 0, 1 0, 1 1)>
>>> list(lines.boundary.geoms)
[<POINT (-1 0)>, <POINT (0 0)>, <POINT (1 0)>, <POINT (1 1)>]
>>> lines.boundary.boundary
<GEOMETRYCOLLECTION EMPTY>
```

See the figures in *LineStrings* and *Collections of Lines* for the illustration of lines and their boundaries.

`object.`**centroid**

Returns a representation of the object's geometric centroid (point).

```
>>> LineString([(0, 0), (1, 1)]).centroid
<POINT (0.5 0.5)>
```

Note: The centroid of an object might be one of its points, but this is not guaranteed.

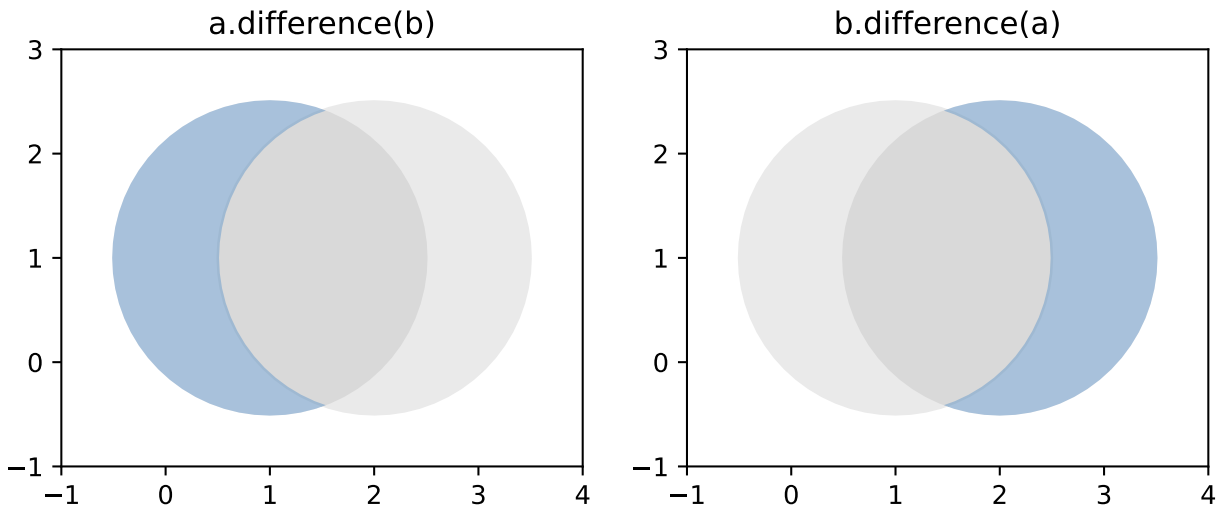
`object.`**difference**(*other*)

Returns a representation of the points making up this geometric object that do not make up the *other* object.

```
>>> a = Point(1, 1).buffer(1.5)
>>> b = Point(2, 1).buffer(1.5)
>>> a.difference(b)
<POLYGON ((1.435 -0.435, 1.293 -0.471, 1.147 -0.493, 1 -0.5, 0.853 -0.493, 0...>
```

Note: The *buffer()* method is used to produce approximately circular polygons in the examples of this section; it will be explained in detail later in this manual.

Figure 8. Differences between two approximately circular polygons.



Note: Shapely can not represent the difference between an object and a lower dimensional object (such as the difference between a polygon and a line or point) as a single object, and in these cases the difference method returns a copy of the object named `self`.

`object.`**`intersection`**`(`*other*`)`

Returns a representation of the intersection of this object with the *other* geometric object.

```
>>> a = Point(1, 1).buffer(1.5)
>>> b = Point(2, 1).buffer(1.5)
>>> a.intersection(b)
<POLYGON ((2.493 0.853, 2.471 0.707, 2.435 0.565, 2.386 0.426, 2.323 0.293, ...>
```

See the figure under *symmetric_difference()* below.

`object.`**`symmetric_difference`**`(`*other*`)`

Returns a representation of the points in this object not in the *other* geometric object, and the points in the *other* not in this geometric object.

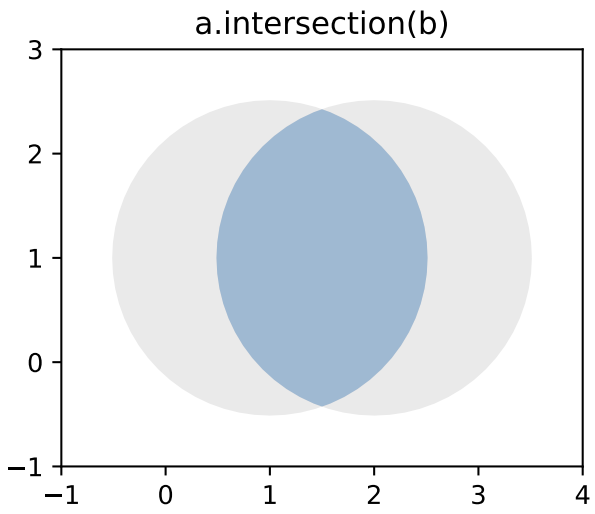
```
>>> a = Point(1, 1).buffer(1.5)
>>> b = Point(2, 1).buffer(1.5)
>>> a.symmetric_difference(b)
<MULTIPOLYGON (((1.574 -0.386, 1.707 -0.323, 1.833 -0.247, 1.952 -0.16, 2.06...>
```

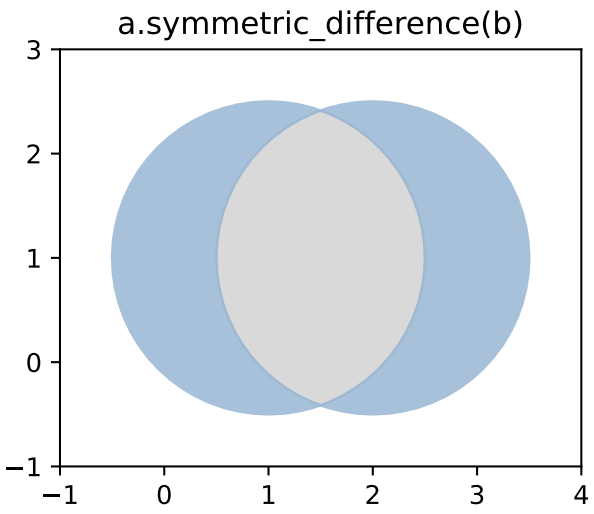
`object.`**`union`**`(`*other*`)`

Returns a representation of the union of points from this object and the *other* geometric object.

The type of object returned depends on the relationship between the operands. The union of polygons (for example) will be a polygon or a multi-polygon depending on whether they intersect or not.

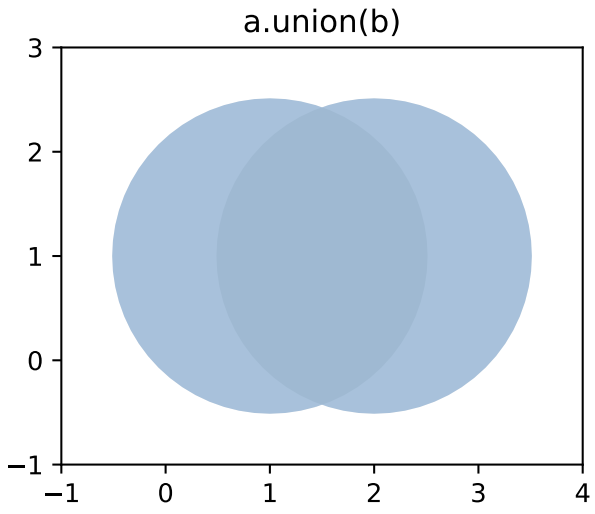
```
>>> a = Point(1, 1).buffer(1.5)
>>> b = Point(2, 1).buffer(1.5)
>>> a.union(b)
<POLYGON ((1.435 -0.435, 1.293 -0.471, 1.147 -0.493, 1 -0.5, 0.853 -0.493, 0...>
```

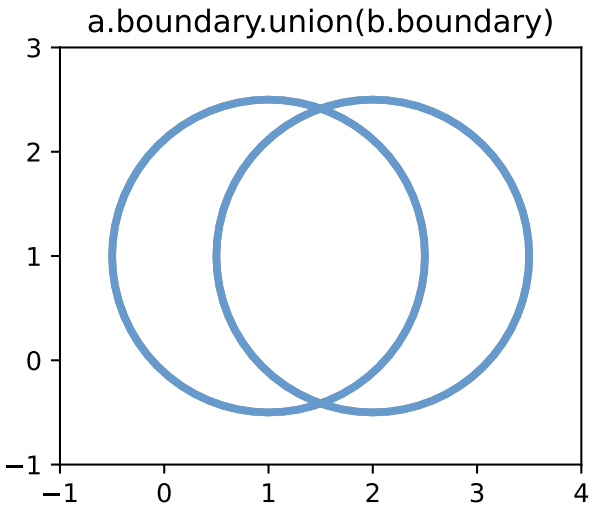




The semantics of these operations vary with type of geometric object. For example, compare the boundary of the union of polygons to the union of their boundaries.

```
>>> a.union(b).boundary
<LINESTRING (1.435 -0.435, 1.293 -0.471, 1.147 -0.493, 1 -0.5, 0.853 -0.493,...>
>>> a.boundary.union(b.boundary)
<MULTILINESTRING ((2.5 1, 2.493 0.853, 2.471 0.707, 2.435 0.565, 2.386 0.426...>
```





Note: *union()* is an expensive way to find the cumulative union of many objects. See *shapely.unary_union()* for a more effective method.

Several of these set-theoretic methods can be invoked using overloaded operators:

- *intersection* can be accessed with and, *&*
- *union* can be accessed with or, |
- *difference* can be accessed with minus, -

- *symmetric_difference* can be accessed with xor, ^

```
>>> from shapely import wkt
>>> p1 = wkt.loads('POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))')
>>> p2 = wkt.loads('POLYGON((0.5 0, 1.5 0, 1.5 1, 0.5 1, 0.5 0))')
>>> p1 & p2
<POLYGON ((0.5 0, 0.5 1, 1 1, 1 0, 0.5 0))>
>>> p1 | p2
<POLYGON ((0 0, 0 1, 0.5 1, 1 1, 1.5 1, 1.5 0, 1 0, 0.5 0, 0 0))>
>>> p1 - p2
<POLYGON ((0 0, 0 1, 0.5 1, 0.5 0, 0 0))>
>>> (p1 ^ p2).wkt
'MULTIPOLYGON (((0 0, 0 1, 0.5 1, 0.5 0, 0 0)), ((1 1, 1.5 1, 1.5 0, 1 0, 1 1)))'
```

Constructive Methods

Shapely geometric object have several methods that yield new objects not derived from set-theoretic analysis.

`object.`**`buffer`**(*distance*, *quad_segs=16*, *cap_style=1*, *join_style=1*, *mitre_limit=5.0*, *single_sided=False*)

Returns an approximate representation of all points within a given *distance* of the this geometric object.

The styles of caps are specified by integer values: 1 (round), 2 (flat), 3 (square). These values are also enumerated by the object *shapely.BufferCapStyle* (see below).

The styles of joins between offset segments are specified by integer values: 1 (round), 2 (mitre), and 3 (bevel). These values are also enumerated by the object *shapely.BufferJoinStyle* (see below).

`shapely.`**`BufferCapStyle`**

Attribute	Value
round	1
flat	2
square	3

`shapely.`**`BufferJoinStyle`**

Attribute	Value
round	1
mitre	2
bevel	3

```
>>> from shapely import BufferCapStyle, BufferJoinStyle
>>> BufferCapStyle.flat.value
2
>>> BufferJoinStyle.bevel.value
3
```

A positive distance has an effect of dilation; a negative distance, erosion. The optional *quad_segs* argument determines the number of segments used to approximate a quarter circle around a point.

```
>>> line = LineString([(0, 0), (1, 1), (0, 2), (2, 2), (3, 1), (1, 0)])
>>> dilated = line.buffer(0.5)
>>> eroded = dilated.buffer(-0.3)
```

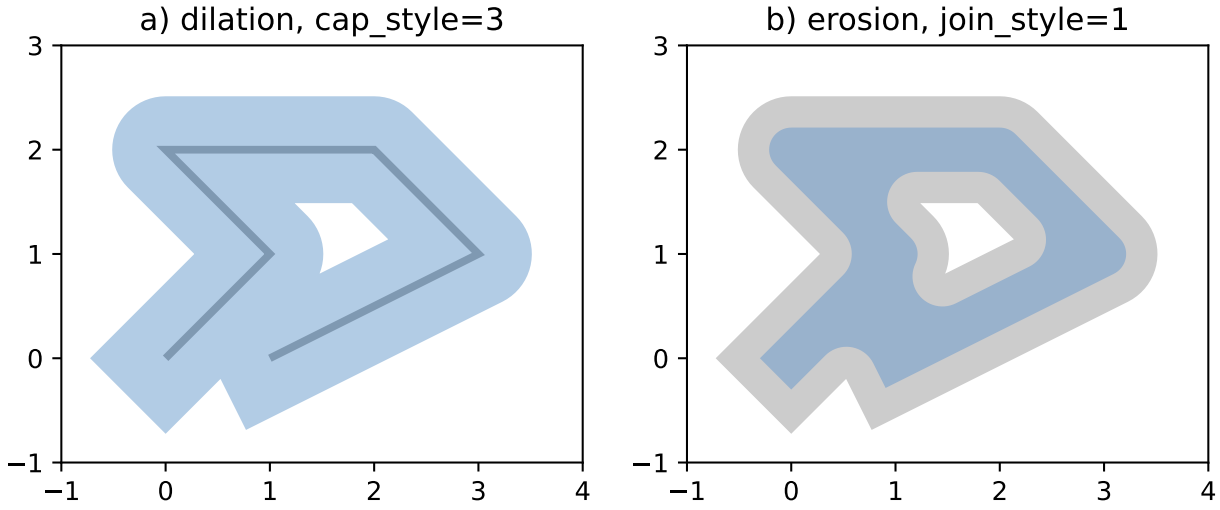


Figure 9. Dilation of a line (left) and erosion of a polygon (right). New object is shown in blue.

The default (*quad_segs* of 16) buffer of a point is a polygonal patch with 99.8% of the area of the circular disk it approximates.

```
>>> p = Point(0, 0).buffer(10.0)
>>> len(p.exterior.coords)
65
>>> p.area
313.6548490545941
```

With a *quad_segs* of 1, the buffer is a square patch.

```
>>> q = Point(0, 0).buffer(10.0, 1)
>>> len(q.exterior.coords)
5
>>> q.area
200.0
```

You may want a buffer only on one side. You can achieve this effect with *single_sided* option.

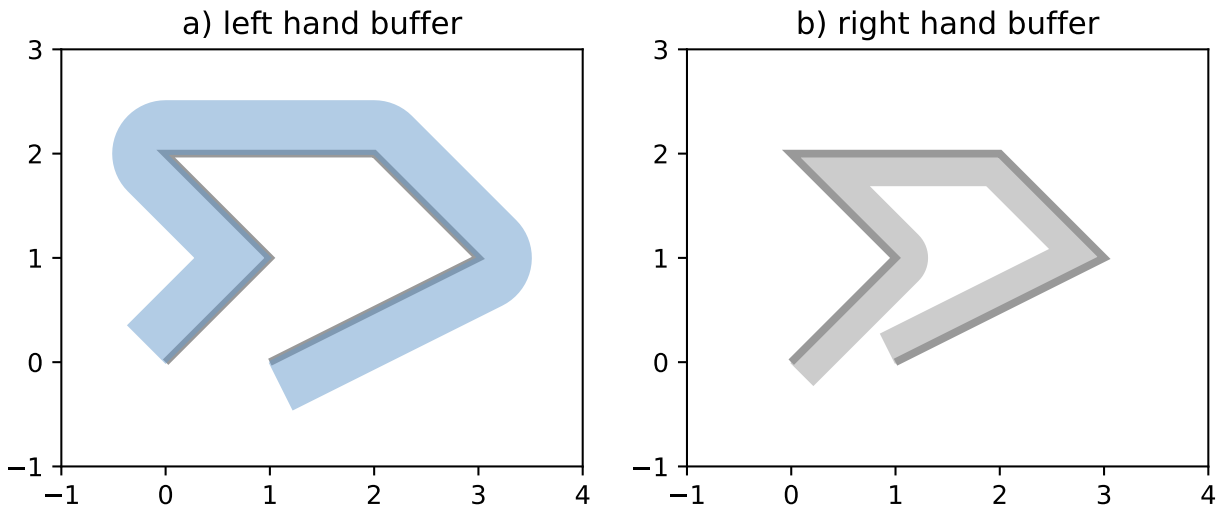
The side used is determined by the sign of the buffer distance:

- a positive distance indicates the left-hand side
- a negative distance indicates the right-hand side

```
>>> line = LineString([(0, 0), (1, 1), (0, 2), (2, 2), (3, 1), (1, 0)])
>>> left_hand_side = line.buffer(0.5, single_sided=True)
>>> right_hand_side = line.buffer(-0.3, single_sided=True)
```

Figure 10. Single sided buffer of 0.5 left hand (left) and of 0.3 right hand (right).

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of *BufferCapStyle.flat*.



Passed a *distance* of 0, *buffer()* can sometimes be used to "clean" self-touching or self-crossing polygons such as the classic "bowtie". Users have reported that very small distance values sometimes produce cleaner results than 0. Your mileage may vary when cleaning surfaces.

```
>>> coords = [(0, 0), (0, 2), (1, 1), (2, 2), (2, 0), (1, 1), (0, 0)]
>>> bowtie = Polygon(coords)
>>> bowtie.is_valid
False
>>> clean = bowtie.buffer(0)
>>> clean.is_valid
True
>>> clean
<MULTIPOLYGON (((0 0, 0 2, 1 1, 0 0)), ((1 1, 2 2, 2 0, 1 1)))>
>>> len(clean.geoms)
2
>>> list(clean.geoms[0].exterior.coords)
[(0.0, 0.0), (0.0, 2.0), (1.0, 1.0), (0.0, 0.0)]
>>> list(clean.geoms[1].exterior.coords)
[(1.0, 1.0), (2.0, 2.0), (2.0, 0.0), (1.0, 1.0)]
```

Buffering splits the polygon in two at the point where they touch.

`object.`**convex_hull**

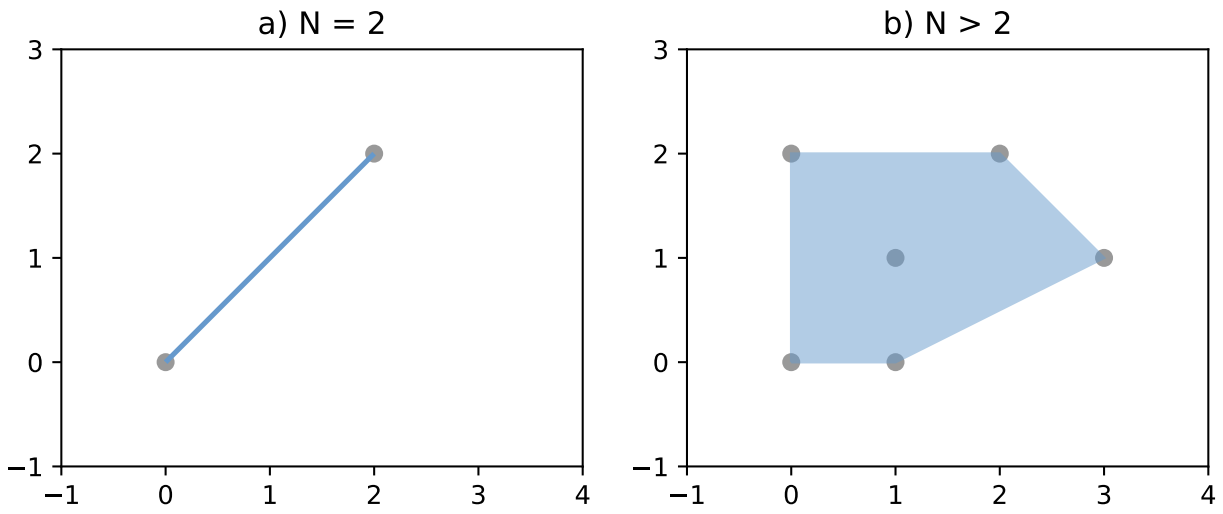
Returns a representation of the smallest convex *Polygon* containing all the points in the object unless the number of points in the object is less than three. For two points, the convex hull collapses to a *LineString*; for 1, a *Point*.

```
>>> Point(0, 0).convex_hull
<POINT (0 0)>
>>> MultiPoint([(0, 0), (1, 1)]).convex_hull
<LINESTRING (0 0, 1 1)>
>>> MultiPoint([(0, 0), (1, 1), (1, -1)]).convex_hull
<POLYGON ((1 -1, 0 0, 1 1, 1 -1))>
```

Figure 11. Convex hull (blue) of 2 points (left) and of 6 points (right).

`object.`**envelope**

Returns a representation of the point or smallest rectangular polygon (with sides parallel to the coordinate axes)



that contains the object.

```
>>> Point(0, 0).envelope
<POINT (0 0)>
>>> MultiPoint([(0, 0), (1, 1)]).envelope
<POLYGON ((0 0, 1 0, 1 1, 0 1, 0 0))>
```

`object.`**`minimum_rotated_rectangle`**

Returns the general minimum bounding rectangle that contains the object. Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

New in Shapely 1.6.0

```
>>> Point(0, 0).minimum_rotated_rectangle
<POINT (0 0)>
>>> MultiPoint([(0,0),(1,1),(2,0.5)]).minimum_rotated_rectangle
<POLYGON ((1.824 1.206, -0.176 0.706, 0 0, 2 0.5, 1.824 1.206))>
```

Figure 12. Minimum rotated rectangle for a multipoint feature (left) and a linestring feature (right).

`object.`**`parallel_offset`**(*distance*, *side*, *resolution=16*, *join_style=1*, *mitre_limit=5.0*)

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

Older alternative method to the *`offset_curve()`* method, but uses *resolution* instead of *quad_segs* and a *side* keyword ('left' or 'right') instead of sign of the distance. This method is kept for backwards compatibility for now, but is is recommended to use *`offset_curve()`* instead.

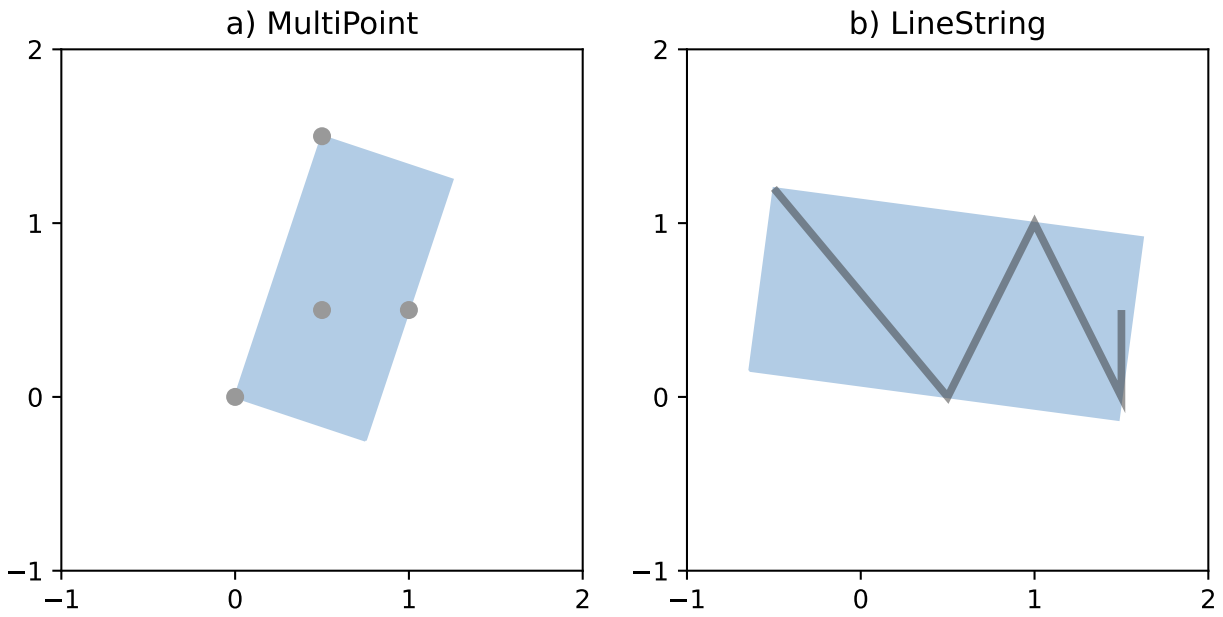
`object.`**`offset_curve`**(*distance*, *quad_segs=16*, *join_style=1*, *mitre_limit=5.0*)

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

The *distance* parameter must be a float value.

The side is determined by the sign of the *distance* parameter (negative for right side offset, positive for left side offset). Left and right are determined by following the direction of the given geometric points of the LineString.

Note: the behaviour regarding orientation of the resulting line depends on the GEOS version. With GEOS < 3.11, the line retains the same direction for a left offset (positive distance) or has reverse direction for a right



offset (negative distance), and this behaviour was documented as such in previous Shapely versions. Starting with GEOS 3.11, the function tries to preserve the orientation of the original line.

The resolution of the offset around each vertex of the object is parameterized as in the *buffer()* method (using *quad_segs*).

The *join_style* is for outside corners between line segments. Accepted integer values are 1 (round), 2 (mitre), and 3 (bevel). See also *shapely.BufferJoinStyle*.

Severely mitered corners can be controlled by the *mitre_limit* parameter (spelled in British English, en-gb). The corners of a parallel line will be further from the original than most places with the mitre join style. The ratio of this further distance to the specified *distance* is the miter ratio. Corners with a ratio which exceed the limit will be beveled.

Note: This method may sometimes return a *MultiLineString* where a simple *LineString* was expected; for example, an offset to a slightly curved LineString.

Note: This method is only available for *LinearRing* and *LineString* objects.

Figure 13. Three styles of parallel offset lines on the left side of a simple line string (its starting point shown as a circle) and one offset on the right side, a multipart.

The effect of the *mitre_limit* parameter is shown below.

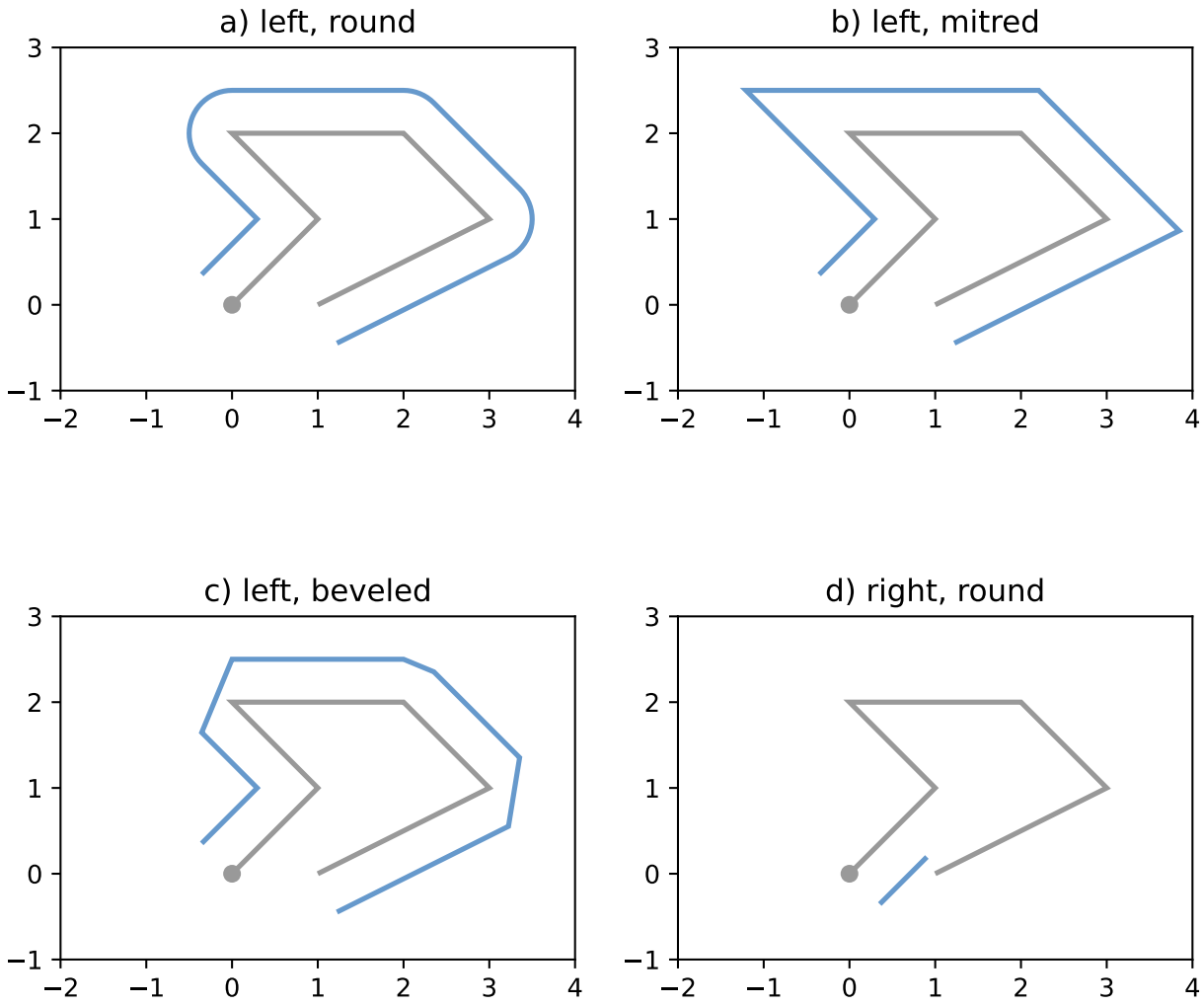
Figure 14. Large and small mitre_limit values for left and right offsets.

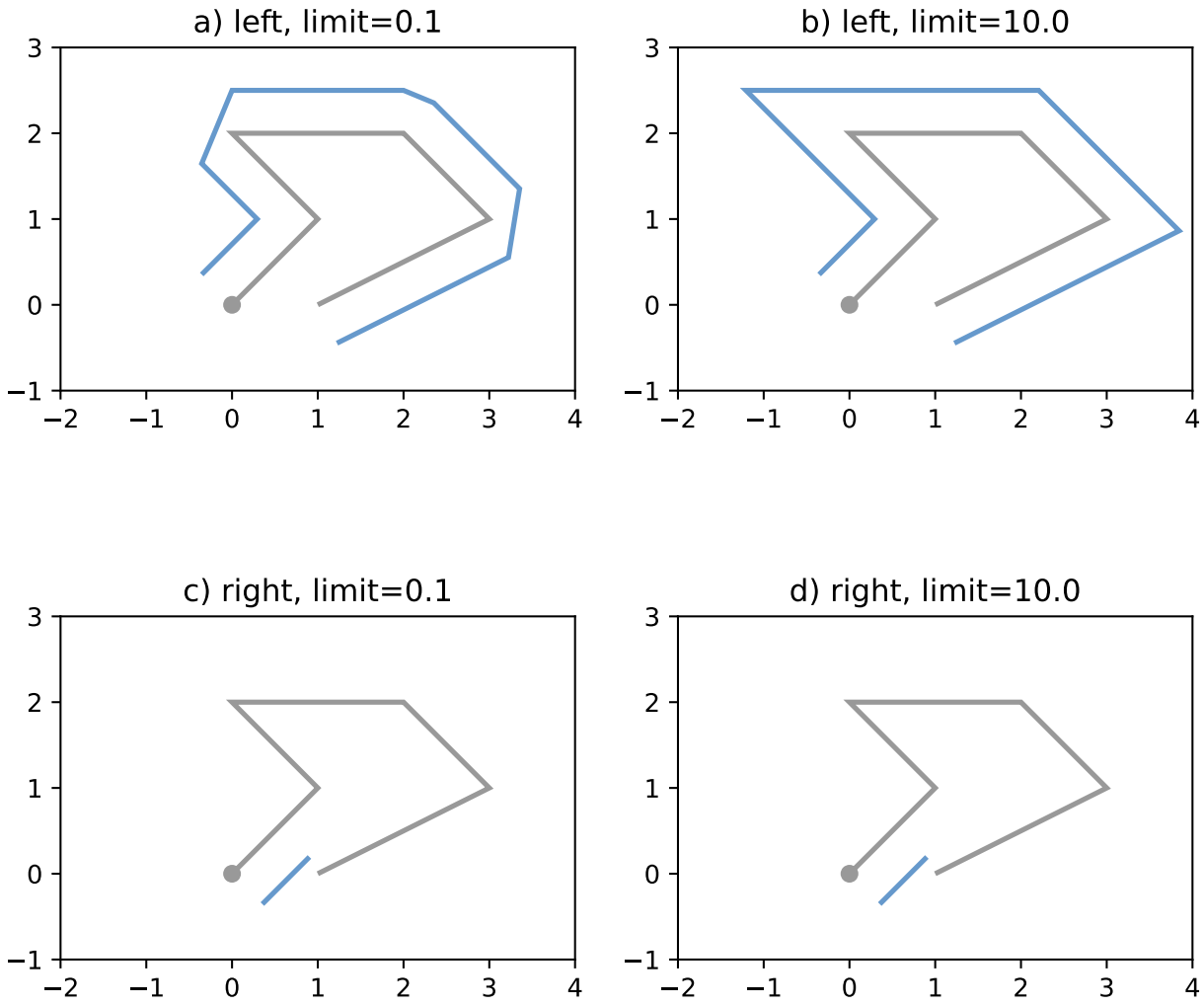
`object.`**`simplify`**(*tolerance*, *preserve_topology=True*)

Returns a simplified representation of the geometric object.

All points in the simplified object will be within the *tolerance* distance of the original geometry. By default a slower algorithm is used that preserves topology. If preserve topology is set to `False` the much quicker Douglas-Peucker algorithm[6] is used.

[6] David H. Douglas and Thomas K. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or


a) left, round
b) left, mitred
c) left, beveled
d) right, round


a) left, limit=0.1
b) left, limit=10.0
c) right, limit=0.1
d) right, limit=10.0

```
>>> p = Point(0.0, 0.0)
>>> x = p.buffer(1.0)
>>> x.area
3.1365484905459398
>>> len(x.exterior.coords)
65
>>> s = x.simplify(0.05, preserve_topology=False)
>>> s.area
3.061467458920719
>>> len(s.exterior.coords)
17
```

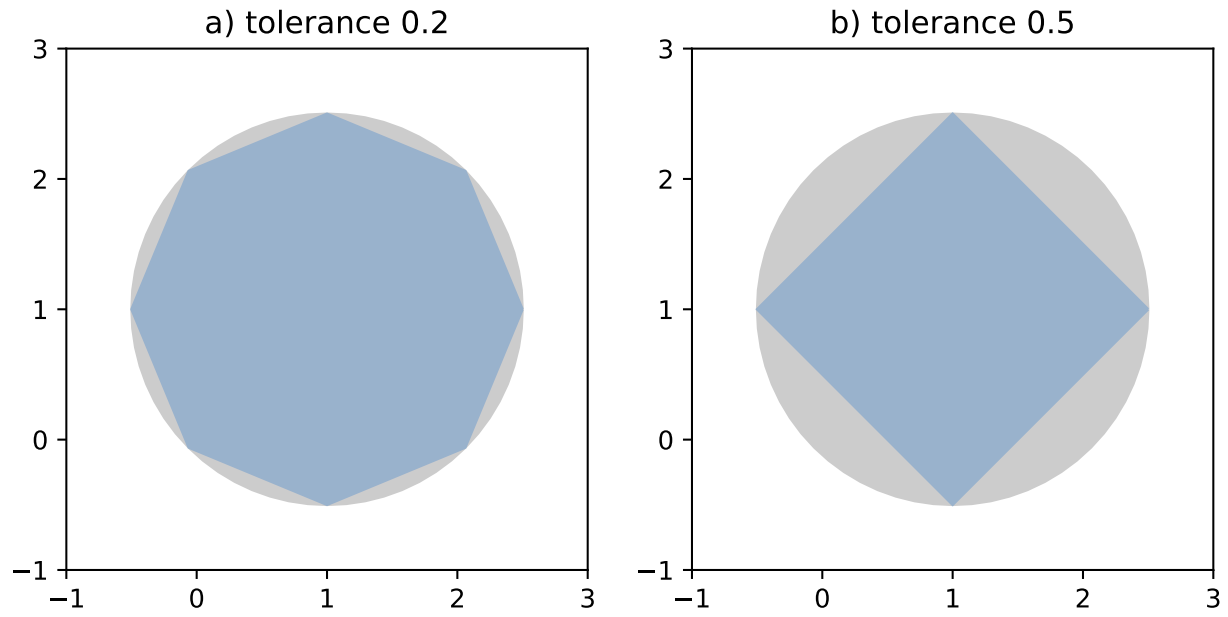


Figure 15. Simplification of a nearly circular polygon using a tolerance of 0.2 (left) and 0.5 (right).

Note: *Invalid* geometric objects may result from simplification that does not preserve topology and simplification may be sensitive to the order of coordinates: two geometries differing only in order of coordinates may be simplified differently.

5.2.5 Affine Transformations

A collection of affine transform functions are in the `shapely.affinity` module, which return transformed geometries by either directly supplying coefficients to an affine transformation matrix, or by using a specific, named transform (*rotate*, *scale*, etc.). The functions can be used with all geometry types (except *GeometryCollection*), and 3D types are either preserved or supported by 3D affine transformations.

New in version 1.2.17.

`shapely.affinity.`**`affine_transform`**`(`*geom*, *matrix*`)`

Returns a transformed geometry using an affine transformation matrix.

its Caricature," Cartographica: The International Journal for Geographic Information and Geovisualization, vol. 10, Dec. 1973, pp. 112-122.

The coefficient `matrix` is provided as a list or tuple with 6 or 12 items for 2D or 3D transformations, respectively.

For 2D affine transformations, the 6 parameter `matrix` is:

```
[a, b, d, e, xoff, yoff]
```

which represents the augmented matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & x_{\text{off}} \\ d & e & y_{\text{off}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

or the equations for the transformed coordinates:

$$x' = ax + by + x_{\text{off}}$$
$$y' = dx + ey + y_{\text{off}}.$$

For 3D affine transformations, the 12 parameter `matrix` is:

```
[a, b, c, d, e, f, g, h, i, xoff, yoff, zoff]
```

which represents the augmented matrix:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & x_{\text{off}} \\ d & e & f & y_{\text{off}} \\ g & h & i & z_{\text{off}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

or the equations for the transformed coordinates:

$$x' = ax + by + cz + x_{\text{off}}$$
$$y' = dx + ey + fz + y_{\text{off}}$$
$$z' = gx + hy + iz + z_{\text{off}}.$$

`shapely.affinity.`**`rotate`**(*geom*, *angle*, *origin='center'*, *use_radians=False*)

Returns a rotated geometry on a 2D plane.

The angle of rotation can be specified in either degrees (default) or radians by setting `use_radians=True`. Positive angles are counter-clockwise and negative are clockwise rotations.

The point of origin can be a keyword `'center'` for the bounding box center (default), `'centroid'` for the geometry's centroid, a *Point* object or a coordinate tuple `(x0, y0)`.

The affine transformation matrix for 2D rotation with angle θ is:

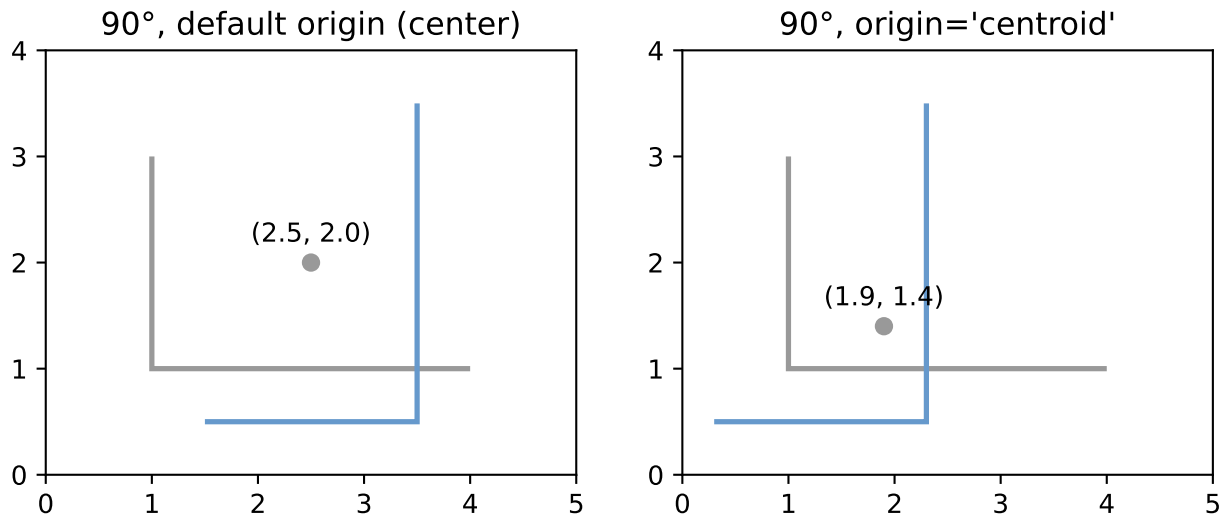
$$\begin{bmatrix} \cos\theta & -\sin\theta & x_{\text{off}} \\ \sin\theta & \cos\theta & y_{\text{off}} \\ 0 & 0 & 1 \end{bmatrix}$$

where the offsets are calculated from the origin (x_0, y_0):

$$x_{\text{off}} = x_0 - x_0 \cos\theta + y_0 \sin\theta$$
$$y_{\text{off}} = y_0 - x_0 \sin\theta - y_0 \cos\theta$$

```
>>> from shapely import affinity
>>> line = LineString([(1, 3), (1, 1), (4, 1)])
>>> rotated_a = affinity.rotate(line, 90)
>>> rotated_b = affinity.rotate(line, 90, origin='centroid')
```

Figure 16. Rotation of a *LineString* (gray) by an angle of 90° counter-clockwise (blue) using different origins.



shapely.affinity.**scale**(*geom*, *xfact=1.0*, *yfact=1.0*, *zfact=1.0*, *origin='center'*)

Returns a scaled geometry, scaled by factors along each dimension.

The point of origin can be a keyword `'center'` for the 2D bounding box center (default), `'centroid'` for the geometry's 2D centroid, a *Point* object or a coordinate tuple `(x0, y0, z0)`.

Negative scale factors will mirror or reflect coordinates.

The general 3D affine transformation matrix for scaling is:

$$\begin{bmatrix} x_{\text{fact}} & 0 & 0 & x_{\text{off}} \\ 0 & y_{\text{fact}} & 0 & y_{\text{off}} \\ 0 & 0 & z_{\text{fact}} & z_{\text{off}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the offsets are calculated from the origin (x_0, y_0, z_0):

$$\begin{aligned} x_{\text{off}} &= x_0 - x_0 x_{\text{fact}} \\ y_{\text{off}} &= y_0 - y_0 y_{\text{fact}} \\ z_{\text{off}} &= z_0 - z_0 z_{\text{fact}} \end{aligned}$$

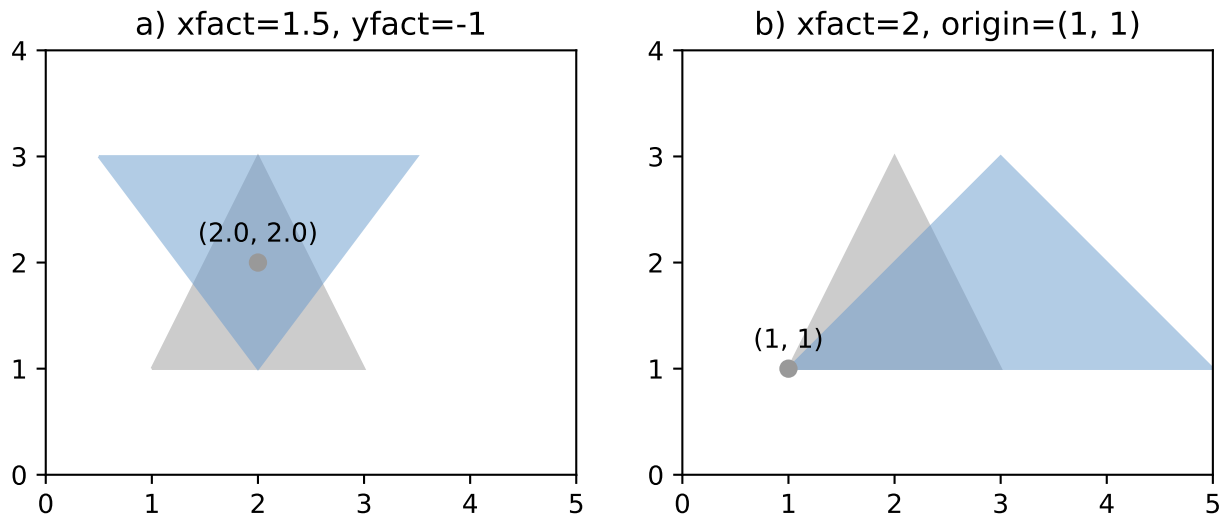
```
>>> triangle = Polygon([(1, 1), (2, 3), (3, 1)])
>>> triangle_a = affinity.scale(triangle, xfact=1.5, yfact=-1)
>>> triangle_a.exterior.coords[:]
[(0.5, 3.0), (2.0, 1.0), (3.5, 3.0), (0.5, 3.0)]
>>> triangle_b = affinity.scale(triangle, xfact=2, origin=(1,1))
>>> triangle_b.exterior.coords[:]
[(1.0, 1.0), (3.0, 3.0), (5.0, 1.0), (1.0, 1.0)]
```

Figure 17. Scaling of a gray triangle to blue result: a) by a factor of 1.5 along x-direction, with reflection across y-axis; b) by a factor of 2 along x-direction with custom origin at (1, 1).

shapely.affinity.**skew**(*geom*, *xs=0.0*, *ys=0.0*, *origin='center'*, *use_radians=False*)

Returns a skewed geometry, sheared by angles along x and y dimensions.

The shear angle can be specified in either degrees (default) or radians by setting `use_radians=True`.



The point of origin can be a keyword `'center'` for the bounding box center (default), `'centroid'` for the geometry's centroid, a *Point* object or a coordinate tuple `(x0, y0)`.

The general 2D affine transformation matrix for skewing is:

$$\begin{bmatrix} 1 & \tan x_s & x_{\text{off}} \\ \tan y_s & 1 & y_{\text{off}} \\ 0 & 0 & 1 \end{bmatrix}$$

where the offsets are calculated from the origin (x_0, y_0):

$$x_{\text{off}} = -y_0 \tan x_s$$
$$y_{\text{off}} = -x_0 \tan y_s$$

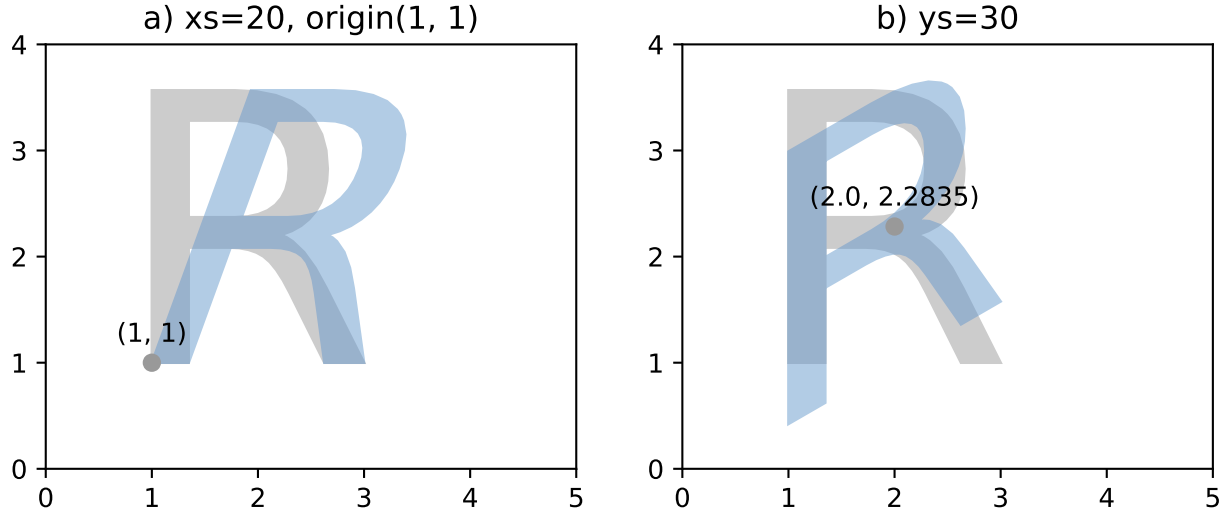


Figure 18. Skewing of a gray "R" to blue result: a) by a shear angle of 20° along the x-direction and an origin at (1, 1); b) by a shear angle of 30° along the y-direction, using default origin.

shapely.affinity.**translate**(*geom*, *xoff=0.0*, *yoff=0.0*, *zoff=0.0*)

Returns a translated geometry shifted by offsets along each dimension.

The general 3D affine transformation matrix for translation is:

$$\begin{bmatrix} 1 & 0 & 0 & x_{\text{off}} \\ 0 & 1 & 0 & y_{\text{off}} \\ 0 & 0 & 1 & z_{\text{off}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.2.6 Other Transformations

Shapely supports map projections and other arbitrary transformations of geometric objects.

shapely.ops.**transform**(*func*, *geom*)

Applies *func* to all coordinates of *geom* and returns a new geometry of the same type from the transformed coordinates.

func maps x, y, and optionally z to output xp, yp, zp. The input parameters may be iterable types like lists or arrays or single values. The output shall be of the same type: scalars in, scalars out; lists in, lists out.

transform tries to determine which kind of function was passed in by calling *func* first with n iterables of coordinates, where n is the dimensionality of the input geometry. If *func* raises a *TypeError* when called with iterables as arguments, then it will instead call *func* on each individual coordinate in the geometry.

New in version 1.2.18.

For example, here is an identity function applicable to both types of input (scalar or array).

```
def id_func(x, y, z=None):
    return tuple(filter(None, [x, y, z]))

g2 = transform(id_func, g1)
```

If using *pyproj*>=*2.1.0*, the preferred method to project geometries is:

```
import pyproj

from shapely import Point
from shapely.ops import transform

wgs84_pt = Point(-72.2495, 43.886)

wgs84 = pyproj.CRS('EPSG:4326')
utm = pyproj.CRS('EPSG:32618')

project = pyproj.Transformer.from_crs(wgs84, utm, always_xy=True).transform
utm_point = transform(project, wgs84_pt)
```

It is important to note that in the example above, the *always_xy* kwarg is required as Shapely only supports coordinates in X,Y order, and in PROJ 6 the WGS84 CRS uses the EPSG-defined Lat/Lon coordinate order instead of the expected Lon/Lat.

If using *pyproj* < *2.1*, then the canonical example is:

```
from functools import partial
import pyproj

from shapely.ops import transform

wgs84 = pyproj.Proj(init='epsg:4326')
utm = pyproj.Proj(init='epsg:32618')

project = partial(
    pyproj.transform,
    wgs84,
    utm)

utm_point = transform(project, wgs84_pt)
```

Lambda expressions such as the one in

```
g2 = transform(lambda x, y, z=None: (x+1.0, y+1.0), g1)
```

also satisfy the requirements for *func*.

5.2.7 Other Operations

Merging Linear Features

Sequences of touching lines can be merged into *MultiLineStrings* or *Polygons* using functions in the `shapely.ops` module.

`shapely.ops.`**`polygonize`**(*lines*)

Returns an iterator over polygons constructed from the input *lines*.

As with the *`MultiLineString`* constructor, the input elements may be any line-like object.

```
>>> from shapely.ops import polygonize
>>> lines = [
...     ((0, 0), (1, 1)),
...     ((0, 0), (0, 1)),
...     ((0, 1), (1, 1)),
...     ((1, 1), (1, 0)),
...     ((1, 0), (0, 0))
...     ]
>>> list(polygonize(lines))
[<POLYGON ((0 0, 1 1, 1 0, 0 0))>, <POLYGON ((1 1, 0 0, 0 1, 1 1))>]
```

`shapely.ops.`**`polygonize_full`**(*lines*)

Creates polygons from a source of lines, returning the polygons and leftover geometries.

The source may be a MultiLineString, a sequence of LineString objects, or a sequence of objects than can be adapted to LineStrings.

Returns a tuple of objects: (polygons, cut edges, dangles, invalid ring lines). Each are a geometry collection.

Dangles are edges which have one or both ends which are not incident on another edge endpoint. Cut edges are connected at both ends but do not form part of polygon. Invalid ring lines form rings which are invalid (bowties, etc).

New in version 1.2.18.

```
>>> from shapely.ops import polygonize_full
>>> lines = [
...     ((0, 0), (1, 1)),
...     ((0, 0), (0, 1)),
...     ((0, 1), (1, 1)),
...     ((1, 1), (1, 0)),
...     ((1, 0), (0, 0)),
...     ((5, 5), (6, 6)),
...     ((1, 1), (100, 100)),
...     ]
>>> result, cuts, dangles, invalids = polygonize_full(lines)
>>> len(result.geoms)
2
>>> list(result.geoms)
[<POLYGON ((0 0, 1 1, 1 0, 0 0))>, <POLYGON ((1 1, 0 0, 0 1, 1 1))>]
>>> list(dangles.geoms)
[<LINESTRING (1 1, 100 100)>, <LINESTRING (5 5, 6 6)>]
```

shapely.ops.**linemerge**(*lines*)

Returns a *LineString* or *MultiLineString* representing the merger of all contiguous elements of *lines*.

As with *shapely.ops.polygonize()*, the input elements may be any line-like object.

```
>>> from shapely.ops import linemerge
>>> linemerge(lines)
<MULTILINESTRING ((1 1, 1 0, 0 0), (0 0, 1 1), (0 0, 0 1, 1 1), (1 1, 100 10...>
>>> list(linemerge(lines).geoms)
[<LINESTRING (1 1, 1 0, 0 0)>,
 <LINESTRING (0 0, 1 1)>,
 <LINESTRING (0 0, 0 1, 1 1)>,
 <LINESTRING (1 1, 100 100)>,
 <LINESTRING (5 5, 6 6)>]
```

Efficient Rectangle Clipping

The *clip_by_rect()* function in *shapely.ops* returns the portion of a geometry within a rectangle.

shapely.ops.**clip_by_rect**(*geom*, *xmin*, *ymin*, *xmax*, *ymax*)

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.

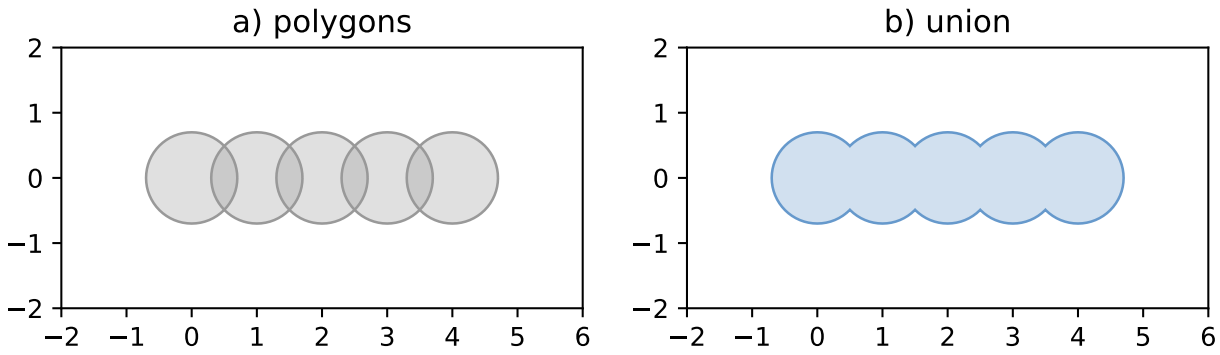
New in version 1.7.

Requires GEOS 3.5.0 or higher

```
>>> from shapely.ops import clip_by_rect
>>> polygon = Polygon(
...     shell=[(0, 0), (0, 30), (30, 30), (30, 0), (0, 0)],
...     holes=[[(10, 10), (20, 10), (20, 20), (10, 20), (10, 10)]],
... )
>>> clipped_polygon = clip_by_rect(polygon, 5, 5, 15, 15)
>>> clipped_polygon
<POLYGON ((5 5, 5 15, 10 15, 10 10, 15 10, 15 5, 5 5))>
```

Efficient Unions

The *unary_union()* function in *shapely.ops* is more efficient than accumulating with *union()*.



shapely.ops.**unary_union**(*geoms*)

Returns a representation of the union of the given geometric objects.

Areas of overlapping *Polygons* will get merged. *LineStrings* will get fully dissolved and noded. Duplicate *Points* will get merged.

```
>>> from shapely.ops import unary_union
>>> polygons = [Point(i, 0).buffer(0.7) for i in range(5)]
>>> unary_union(polygons)
<POLYGON ((0.444 -0.541, 0.389 -0.582, 0.33 -0.617, 0.268 -0.647, 0.203 -0.6...>
```

Because the union merges the areas of overlapping *Polygons* it can be used in an attempt to fix invalid *MultiPolygons*. As with the zero distance *buffer()* trick, your mileage may vary when using this.

```
>>> m = MultiPolygon(polygons)
>>> m.area
7.684543801837549
>>> m.is_valid
False
>>> unary_union(m).area
6.610301355116799
>>> unary_union(m).is_valid
True
```

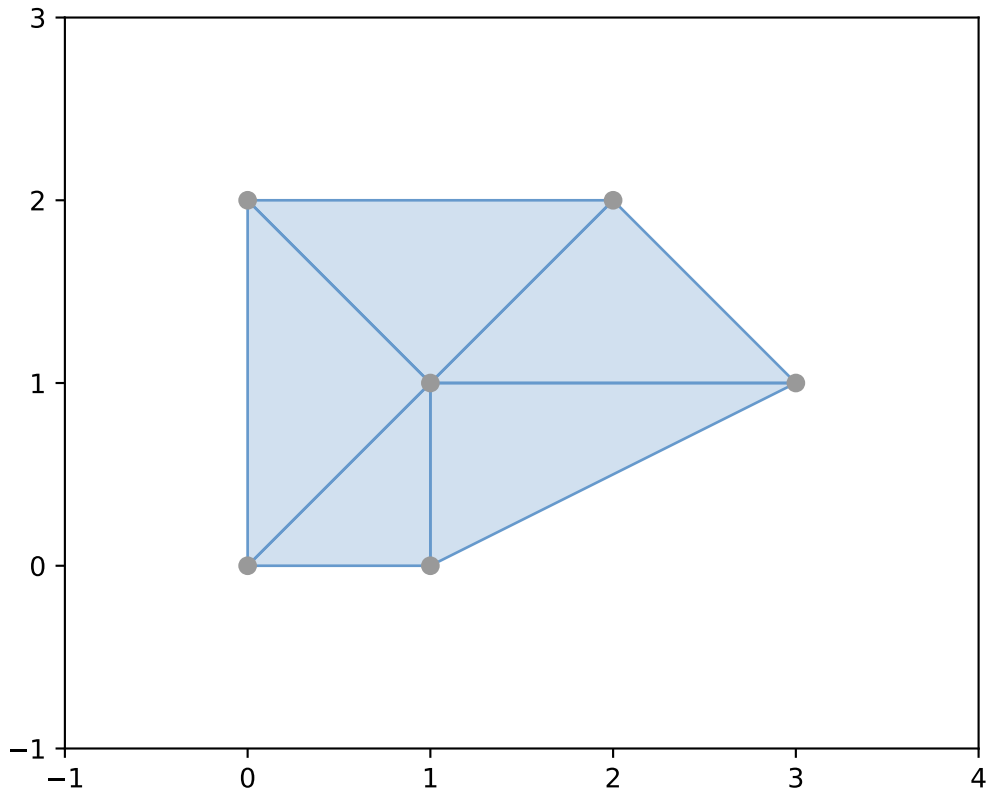
shapely.ops.**cascaded_union**(*geoms*)

Returns a representation of the union of the given geometric objects.

Note: In 1.8.0 *shapely.ops.cascaded_union()* is deprecated, as it was superseded by *shapely.ops.unary_union()*.

Delaunay triangulation

The `triangulate()` function in *shapely.ops* calculates a Delaunay triangulation from a collection of points.



shapely.ops.**triangulate**(*geom*, *tolerance=0.0*, *edges=False*)

Returns a Delaunay triangulation of the vertices of the input geometry.

The source may be any geometry type. All vertices of the geometry will be used as the points of the triangulation.

The *tolerance* keyword argument sets the snapping tolerance used to improve the robustness of the triangulation computation. A tolerance of 0.0 specifies that no snapping will take place.

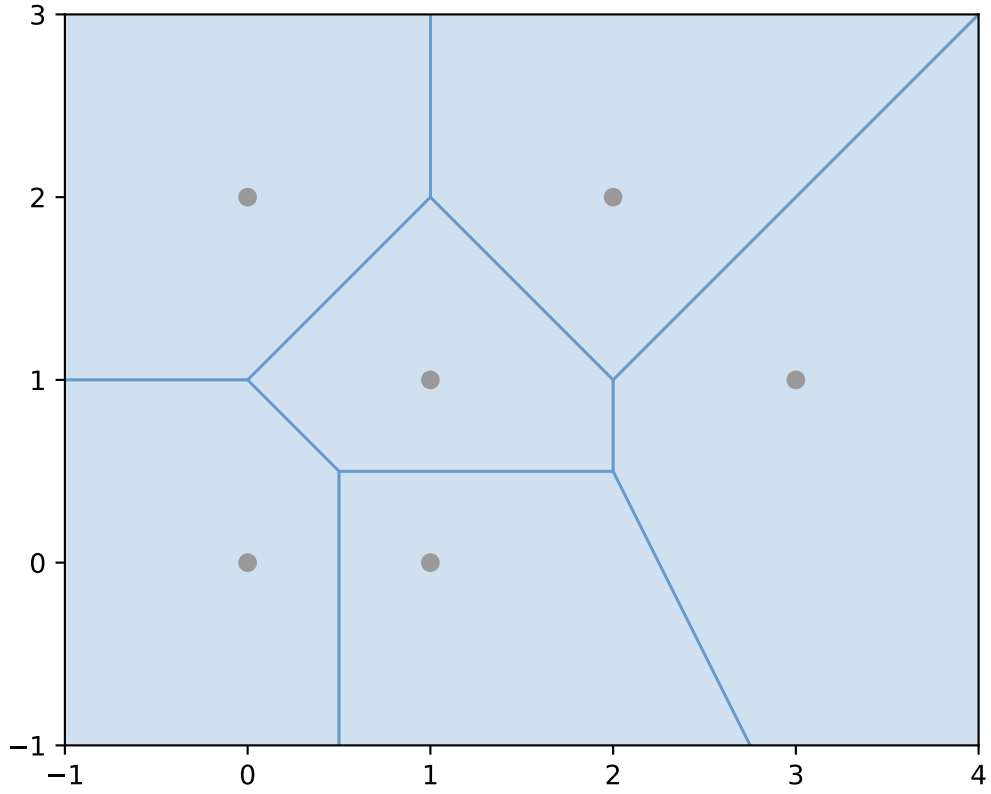
If the *edges* keyword argument is *False* a list of *Polygon* triangles will be returned. Otherwise a list of *LineString* edges is returned.

New in version 1.4.0

```
>>> from shapely.ops import triangulate
>>> points = MultiPoint([(0, 0), (1, 1), (0, 2), (2, 2), (3, 1), (1, 0)])
>>> triangulate(points)
[<POLYGON ((0 2, 0 0, 1 1, 0 2))>,
 <POLYGON ((0 2, 1 1, 2 2, 0 2))>,
 <POLYGON ((2 2, 1 1, 3 1, 2 2))>,
 <POLYGON ((3 1, 1 1, 1 0, 3 1))>,
 <POLYGON ((1 0, 1 1, 0 0, 1 0))>]
```

Voronoi Diagram

The *voronoi_diagram()* function in *shapely.ops* constructs a Voronoi diagram from a collection points, or the vertices of any geometry.



shapely.ops.**voronoi_diagram**(*geom*, *envelope=None*, *tolerance=0.0*, *edges=False*)

Constructs a Voronoi diagram from the vertices of the input geometry.

The source may be any geometry type. All vertices of the geometry will be used as the input points to the diagram.

The *envelope* keyword argument provides an envelope to use to clip the resulting diagram. If *None*, it will be calculated automatically. The diagram will be clipped to the *larger* of the provided envelope or an envelope surrounding the sites.

The *tolerance* keyword argument sets the snapping tolerance used to improve the robustness of the computation. A tolerance of 0.0 specifies that no snapping will take place. The tolerance *argument* can be finicky and is known to cause the algorithm to fail in several cases. If you're using *tolerance* and getting a failure, try removing it. The test cases in *tests/test_voronoi_diagram.py* show more details.

If the *edges* keyword argument is *False* a list of *Polygon`s will be returned. Otherwise a list of `LineString* edges is returned.

```
>>> from shapely.ops import voronoi_diagram
>>> points = MultiPoint([(0, 0), (1, 1), (0, 2), (2, 2), (3, 1), (1, 0)])
>>> regions = voronoi_diagram(points)
>>> list(regions.geoms)
[<POLYGON ((2 1, 2 0.5, 0.5 0.5, 0 1, 1 2, 2 1))>,
 <POLYGON ((6 -3, 3.75 -3, 2 0.5, 2 1, 6 5, 6 -3))>,
```

(continues on next page)

(continued from previous page)

```
 <POLYGON ((-3 -3, -3 1, 0 1, 0.5 0.5, 0.5 -3, -3 -3))>,
 <POLYGON ((0.5 -3, 0.5 0.5, 2 0.5, 3.75 -3, 0.5 -3))>,
 <POLYGON ((-3 5, 1 5, 1 2, 0 1, -3 1, -3 5))>,
 <POLYGON ((6 5, 2 1, 1 2, 1 5, 6 5))>]
```

Nearest points

The *nearest_points()* function in *shapely.ops* calculates the nearest points in a pair of geometries.

shapely.ops.**nearest_points**(*geom1*, *geom2*)

Returns a tuple of the nearest points in the input geometries. The points are returned in the same order as the input geometries.

New in version 1.4.0.

```
>>> from shapely.ops import nearest_points
>>> triangle = Polygon([(0, 0), (1, 0), (0.5, 1), (0, 0)])
>>> square = Polygon([(0, 2), (1, 2), (1, 3), (0, 3), (0, 2)])
>>> list(nearest_points(triangle, square))
[<POINT (0.5 1)>, <POINT (0.5 2)>]
```

Note that the nearest points may not be existing vertices in the geometries.

Snapping

The *snap()* function in *shapely.ops* snaps the vertices in one geometry to the vertices in a second geometry with a given tolerance.

shapely.ops.**snap**(*geom1*, *geom2*, *tolerance*)

Snaps vertices in *geom1* to vertices in the *geom2*. A copy of the snapped geometry is returned. The input geometries are not modified.

The *tolerance* argument specifies the minimum distance between vertices for them to be snapped.

New in version 1.5.0

```
>>> from shapely.ops import snap
>>> square = Polygon([(1,1), (2, 1), (2, 2), (1, 2), (1, 1)])
>>> line = LineString([(0,0), (0.8, 0.8), (1.8, 0.95), (2.6, 0.5)])
>>> result = snap(line, square, 0.5)
>>> result
<LINESTRING (0 0, 1 1, 2 1, 2.6 0.5)>
```

Shared paths

The *shared_paths()* function in *shapely.ops* finds the shared paths between two linear geometries.

shapely.ops.**shared_paths**(*geom1*, *geom2*)

Finds the shared paths between *geom1* and *geom2*, where both geometries are *LineStrings*.

A *GeometryCollection* is returned with two elements. 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.

New in version 1.6.0

```
>>> from shapely.ops import shared_paths
>>> g1 = LineString([(0, 0), (10, 0), (10, 5), (20, 5)])
>>> g2 = LineString([(5, 0), (30, 0), (30, 5), (0, 5)])
>>> forward, backward = shared_paths(g1, g2).geoms
>>> forward
<MULTILINESTRING ((5 0, 10 0))>
>>> backward
<MULTILINESTRING ((10 5, 20 5))>
```

Splitting

The *split()* function in *shapely.ops* splits a geometry by another geometry.

shapely.ops.**split**(*geom*, *splitter*)

Splits a geometry by another geometry and returns a collection of geometries. This function is the theoretical opposite of the union of the split geometry parts. If the splitter does not split the geometry, a collection with a single geometry equal to the input geometry is returned.

The function supports:

- Splitting a (Multi)LineString by a (Multi)Point or (Multi)LineString or (Multi)Polygon boundary
- Splitting a (Multi)Polygon by a LineString

It may be convenient to snap the splitter with low tolerance to the geometry. For example in the case of splitting a line by a point, the point must be exactly on the line, for the line to be correctly split. When splitting a line by a polygon, the boundary of the polygon is used for the operation. When splitting a line by another line, a ValueError is raised if the two overlap at some segment.

New in version 1.6.0

```
>>> from shapely.ops import split
>>> pt = Point((1, 1))
>>> line = LineString([(0,0), (2,2)])
>>> result = split(line, pt)
>>> result
<GEOMETRYCOLLECTION (LINESTRING (0 0, 1 1), LINESTRING (1 1, 2 2))>
```

Substring

The *substring()* function in `shapely.ops` returns a line segment between specified distances along a *LineString*.

`shapely.ops.`**`substring`**(*geom*, *start_dist*, *end_dist*[, *normalized=False*])

Return the *LineString* between *start_dist* and *end_dist* or a *Point* if they are at the same location

Negative distance values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values.

If the start distance equals the end distance, a point is being returned.

If the start distance is actually past the end distance, then the reversed substring is returned such that the start distance is at the first coordinate.

If the normalized arg is `True`, the distance will be interpreted as a fraction of the geometry's length

New in version 1.7.0

Here are some examples that return *LineString* geometries.

```
>>> from shapely.ops import substring
>>> ls = LineString((i, 0) for i in range(6))
>>> ls
<LINESTRING (0 0, 1 0, 2 0, 3 0, 4 0, 5 0)>
>>> substring(ls, start_dist=1, end_dist=3)
<LINESTRING (1 0, 2 0, 3 0)>
>>> substring(ls, start_dist=3, end_dist=1)
<LINESTRING (3 0, 2 0, 1 0)>
>>> substring(ls, start_dist=1, end_dist=-3)
<LINESTRING (1 0, 2 0)>
>>> substring(ls, start_dist=0.2, end_dist=-0.6, normalized=True)
<LINESTRING (1 0, 2 0)>
```

And here is an example that returns a *Point*.

```
>>> substring(ls, start_dist=2.5, end_dist=-2.5)
<POINT (2.5 0)>
```

Prepared Geometry Operations

Shapely geometries can be processed into a state that supports more efficient batches of operations.

`prepared.`**`prep`**(*ob*)

Creates and returns a prepared geometric object.

To test one polygon containment against a large batch of points, one should first use the *prepared.prep()* function.

```
>>> from shapely.prepared import prep
>>> points = [...] # large list of points
>>> polygon = Point(0.0, 0.0).buffer(1.0)
>>> prepared_polygon = prep(polygon)
>>> prepared_polygon
<shapely.prepared.PreparedGeometry object at 0x...>
>>> hits = filter(prepared_polygon.contains, points)
```

Prepared geometries instances have the following methods: `contains`, `contains_properly`, `covers`, and `intersects`. All have exactly the same arguments and usage as their counterparts in non-prepared geometric objects.

Diagnostics

validation.explain_validity(ob):

Returns a string explaining the validity or invalidity of the object.

New in version 1.2.1.

The messages may or may not have a representation of a problem point that can be parsed out.

```
>>> coords = [(0, 0), (0, 2), (1, 1), (2, 2), (2, 0), (1, 1), (0, 0)]
>>> p = Polygon(coords)
>>> from shapely.validation import explain_validity
>>> explain_validity(p)
'Ring Self-intersection[1 1]'
```

validation.**make_valid**(*ob*)

Returns a valid representation of the geometry, if it is invalid. If it is valid, the input geometry will be returned.

In many cases, in order to create a valid geometry, the input geometry must be split into multiple parts or multiple geometries. If the geometry must be split into multiple parts of the same geometry type, then a multi-part geometry (e.g. a MultiPolygon) will be returned. if the geometry must be split into multiple parts of different types, then a GeometryCollection will be returned.

For example, this operation on a geometry with a bow-tie structure:

```
>>> from shapely.validation import make_valid
>>> coords = [(0, 0), (0, 2), (1, 1), (2, 2), (2, 0), (1, 1), (0, 0)]
>>> p = Polygon(coords)
>>> make_valid(p)
<MULTIPOLYGON (((1 1, 0 0, 0 2, 1 1)), ((2 0, 1 1, 2 2, 2 0)))>

Yields a MultiPolygon with two parts:
```

```
>>> from shapely.validation import make_valid
>>> coords = [(0, 2), (0, 1), (2, 0), (0, 0), (0, 2)]
>>> p = Polygon(coords)
>>> make_valid(p)
<GEOMETRYCOLLECTION (POLYGON ((2 0, 0 0, 0 1, 2 0)), LINESTRING (0 2, 0 1))>

Yields a GeometryCollection with a Polygon and a LineString:
```

The Shapely version, GEOS library version, and GEOS C API version are accessible via `shapely.__version__`, `shapely.geos_version_string`, and `shapely.geos_capi_version`.

```
>>> import shapely
>>> shapely.__version__
'2.0.0'
>>> shapely.geos_version
(3, 10, 2)
>>> shapely.geos_capi_version_string
'3.10.2-CAPI-1.16.0'
```

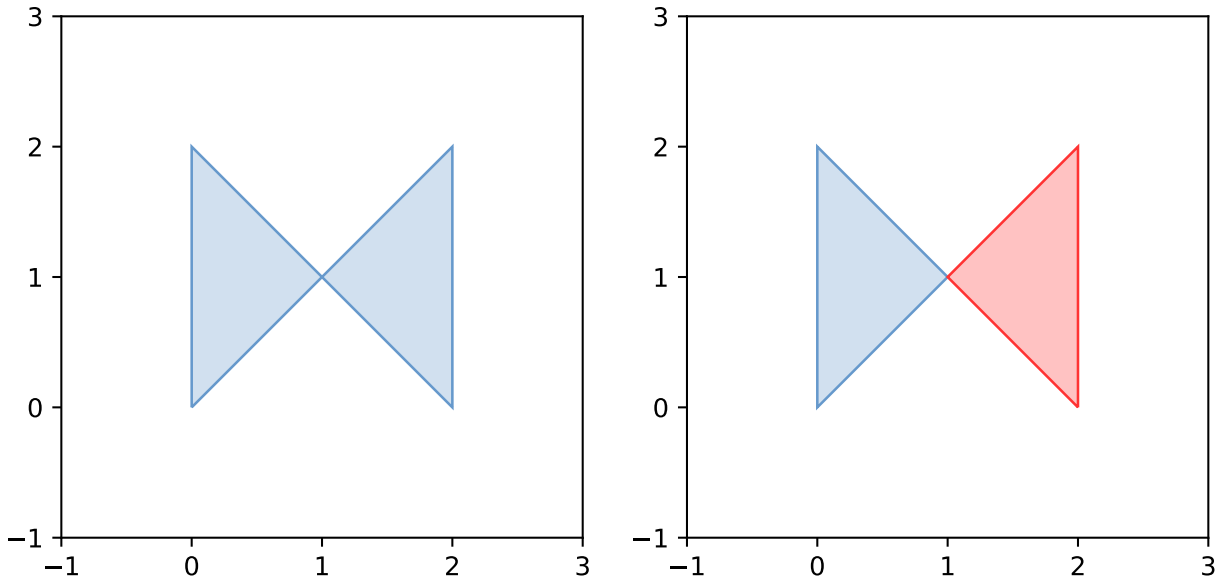


Fig. 1: While this operation:

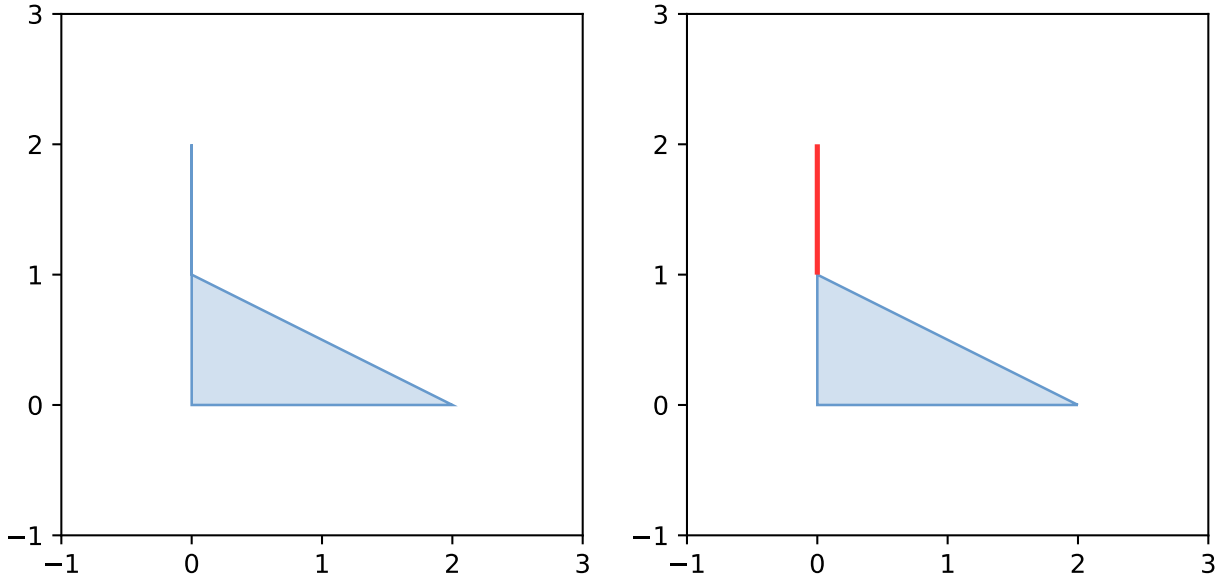


Fig. 2: *New in version 1.8 Requires GEOS > 3.8*

Polylabel

shapely.ops.**polylabel**(*polygon*, *tolerance*)

Finds the approximate location of the pole of inaccessibility for a given polygon. Based on Vladimir Agafonkin's polylabel.

New in version 1.6.0

Note: Prior to 1.7 *polylabel* must be imported from *shapely.algorithms.polylabel* instead of *shapely.ops*.

```
>>> from shapely.ops import polylabel
>>> polygon = LineString([(0, 0), (50, 200), (100, 100), (20, 50),
... (-100, -20), (-150, -200)]).buffer(100)
>>> label = polylabel(polygon, tolerance=10)
>>> label
<POINT (59.356 121.839)>
```

5.2.8 STR-packed R-tree

Shapely provides an interface to the query-only GEOS R-tree packed using the Sort-Tile-Recursive algorithm. Pass a list of geometry objects to the STRtree constructor to create a spatial index that you can query with another geometric object. Query-only means that once created, the *STRtree* is immutable. You cannot add or remove geometries.

class strtree.**STRtree**(*geometries*)

The *STRtree* constructor takes a sequence of geometric objects.

References to these geometric objects are kept and stored in the R-tree.

New in version 1.4.0.

strtree.**query**(*geom*)

Returns the integer indices of all geometries in the *strtree* whose extents intersect the extent of *geom*. This means that a subsequent search through the returned subset using the desired binary predicate (eg. intersects, crosses, contains, overlaps) may be necessary to further filter the results according to their specific spatial relationships.

```
>>> from shapely import STRtree
>>> points = [Point(i, i) for i in range(10)]
>>> tree = STRtree(points)
>>> query_geom = Point(2,2).buffer(0.99)
>>> [points[idx].wkt for idx in tree.query(query_geom)]
['POINT (2 2)']
>>> query_geom = Point(2, 2).buffer(1.0)
>>> [points[idx].wkt for idx in tree.query(query_geom)]
['POINT (1 1)', 'POINT (2 2)', 'POINT (3 3)']
>>> [points[idx].wkt for idx in tree.query(query_geom, predicate="intersects")]
['POINT (2 2)']
```

strtree.**nearest**(*geom*)

Returns the nearest geometry in *strtree* to *geom*.

```
>>> points = [Point(i, i) for i in range(10)]
>>> tree = STRtree(points)
>>> idx = tree.nearest(Point(2.2, 2.2))
>>> points[idx]
<POINT (2 2)>
```

5.2.9 Interoperation

Shapely provides 4 avenues for interoperation with other software.

Well-Known Formats

A *Well Known Text* (WKT) or *Well Known Binary* (WKB) representation[Page 20, 1] of any geometric object can be had via its `wkt` or `wkb` attribute. These representations allow interchange with many GIS programs. PostGIS, for example, trades in hex-encoded WKB.

```
>>> Point(0, 0).wkt
'POINT (0 0)'
>>> Point(0, 0).wkb
b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> Point(0, 0).wkb_hex
'010100000000000000000000000000000000000000'
```

The *shapely.wkt* and *shapely.wkb* modules provide *dumps()* and *loads()* functions that work almost exactly as their *pickle* and *simplejson* module counterparts. To serialize a geometric object to a binary or text string, use `dumps()`. To deserialize a string and get a new geometric object of the appropriate type, use `loads()`.

The default settings for the wkt attribute and *shapely.wkt.dumps()* function are different. By default, the attribute's value is trimmed of excess decimals, while this is not the case for *dumps()*, though it can be replicated by setting *trim=True*.

`shapely.wkb.`**`dumps`**`(`*ob*`)`

Returns a WKB representation of *ob*.

`shapely.wkb.`**`loads`**`(`*wkb*`)`

Returns a geometric object from a WKB representation *wkb*.

```
>>> from shapely import wkb
>>> pt = Point(0, 0)
>>> wkb.dumps(pt)
b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> pt.wkb
b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> wkb.loads(pt.wkb).wkt
'POINT (0 0)'
```

All of Shapely's geometry types are supported by these functions.

`shapely.wkt.`**`dumps`**`(`*ob*`)`

Returns a WKT representation of *ob*. Several keyword arguments are available to alter the WKT which is returned; see the docstrings for more details.

`shapely.wkt.`**`loads`**`(`*wkt*`)`

Returns a geometric object from a WKT representation *wkt*.

```
>>> from shapely import wkt
>>> pt = Point(0, 0)
>>> thewkt = wkt.dumps(pt)
>>> thewkt
'POINT (0.0000000000000000 0.0000000000000000)'
>>> pt.wkt
'POINT (0 0)'
>>> wkt.dumps(pt, trim=True)
'POINT (0 0)'
```

Numpy and Python Arrays

All geometric objects with coordinate sequences (*Point*, *LinearRing*, *LineString*) provide the Numpy array interface and can thereby be converted or adapted to Numpy arrays.

```
>>> import numpy as np
>>> np.asarray(Point(0, 0).coords)
array([[0., 0.]])
>>> np.asarray(LineString([(0, 0), (1, 1)]).coords)
array([[0., 0.],
       [1., 1.]])
```

The coordinates of the same types of geometric objects can be had as standard Python arrays of x and y values via the `xy` attribute.

```
>>> Point(0, 0).xy
(array('d', [0.0]), array('d', [0.0]))
>>> LineString([(0, 0), (1, 1)]).xy
(array('d', [0.0, 1.0]), array('d', [0.0, 1.0]))
```

Python Geo Interface

Any object that provides the GeoJSON-like Python geo interface can be converted to a Shapely geometry using the *shapely.geometry.shape()* function.

shapely.geometry.**shape**(*context*)

Returns a new, independent geometry with coordinates *copied* from the context.

For example, a dictionary:

```
>>> from shapely.geometry import shape
>>> data = {"type": "Point", "coordinates": (0.0, 0.0)}
>>> geom = shape(data)
>>> geom.geom_type
'Point'
>>> list(geom.coords)
[(0.0, 0.0)]
```

Or a simple placemark-type object:

```
>>> class GeoThing:
...     def __init__(self, d):
```

(continues on next page)

(continued from previous page)

```
...         self.__geo_interface__ = d
>>> thing = GeoThing({"type": "Point", "coordinates": (0.0, 0.0)})
>>> geom = shape(thing)
>>> geom.geom_type
'Point'
>>> list(geom.coords)
[(0.0, 0.0)]
```

The GeoJSON-like mapping of a geometric object can be obtained using *shapely.geometry.mapping()*.

shapely.geometry.**mapping**(*ob*)

Returns a GeoJSON-like mapping from a Geometry or any object which implements __geo_interface__.

New in version 1.2.3.

For example, using the same *GeoThing* class:

```
>>> from shapely.geometry import mapping
>>> thing = GeoThing({"type": "Point", "coordinates": (0.0, 0.0)})
>>> m = mapping(thing)
>>> m['type']
'Point'
>>> m['coordinates']
(0.0, 0.0)
```

5.2.10 Performance

Shapely uses the GEOS library for all operations. GEOS is written in C++ and used in many applications and you can expect that all operations are highly optimized. The creation of new geometries with many coordinates, however, involves some overhead that might slow down your code.

5.2.11 Conclusion

We hope that you will enjoy and profit from using Shapely. This manual will be updated and improved regularly. Its source is available at https://github.com/shapely/shapely/tree/main/docs/.

5.2.12 References

5.3 Migrating to Shapely 1.8 / 2.0

Shapely 1.8.0 is a transitional version introducing several warnings in preparation of the upcoming changes in 2.0.0.

Shapely 2.0.0 will be a major release with a refactor of the internals with considerable performance improvements (based on the developments in the PyGEOS package), along with several breaking changes.

This guide gives an overview of the most important changes with details on what will change in 2.0.0, how we warn for this in 1.8.0, and how you can update your code to be future-proof.

For more background, see RFC 1: Roadmap for Shapely 2.0.

5.3.1 Geometry objects will become immutable

Geometry objects will become immutable in version 2.0.0.

In Shapely 1.x, some of the geometry classes are mutable, meaning that you can change their coordinates in-place. Illustrative code:

```
>>> from shapely.geometry import LineString
>>> line = LineString([(0,0), (2, 2)])
>>> print(line)
LINESTRING (0 0, 2 2)

>>> line.coords = [(0, 0), (10, 0), (10, 10)]
>>> print(line)
LINESTRING (0 0, 10 0, 10 10)
```

In Shapely 1.8, this will start raising a warning:

```
>>> line.coords = [(0, 0), (10, 0), (10, 10)]
ShapelyDeprecationWarning: Setting the 'coords' to mutate a Geometry
in place is deprecated, and will not be possible any more in Shapely 2.0
```

and starting with version 2.0.0, all geometry objects will become immutable. As a consequence, they will also become hashable and therefore usable as, for example, dictionary keys.

How do I update my code? There is no direct alternative for mutating the coordinates of an existing geometry, except for creating a new geometry object with the new coordinates.

Setting custom attributes

Another consequence of the geometry objects becoming immutable is that assigning custom attributes, which currently works, will no longer be possible.

Currently you can do:

```
>>> line.name = "my_geometry"
>>> line.name
'my_geometry'
```

In Shapely 1.8, this will start raising a warning, and will raise an AttributeError in Shapely 2.0.

How do I update my code? There is no direct alternative for adding custom attributes to geometry objects. You can use other Python data structures such as (GeoJSON-like) dictionaries or GeoPandas' GeoDataFrames to store attributes alongside geometry features.

5.3.2 Multi-part geometries will no longer be "sequences" (length, iterable, indexable)

In Shapely 1.x, multi-part geometries (MultiPoint, MultiLineString, MultiPolygon and GeometryCollection) implement a part of the "sequence" python interface (making them list-like). This means you can iterate through the object to get the parts, index into the object to get a specific part, and ask for the number of parts with the `len()` method.

Some examples of this with Shapely 1.x:

```
>>> from shapely.geometry import Point, MultiPoint
>>> mp = MultiPoint([(1, 1), (2, 2), (3, 3)])
>>> print(mp)
MULTIPOINT (1 1, 2 2, 3 3)
>>> for part in mp:
...     print(part)
POINT (1 1)
POINT (2 2)
POINT (3 3)
>>> print(mp[1])
POINT (2 2)
>>> len(mp)
3
>>> list(mp)
[<shapely.geometry.point.Point at 0x7f2e0912bf10>,
 <shapely.geometry.point.Point at 0x7f2e09fed820>,
 <shapely.geometry.point.Point at 0x7f2e09fed4c0>]
```

Starting with Shapely 1.8, all the examples above will start raising a deprecation warning. For example:

```
>>> for part in mp:
...     print(part)
ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated
and will be removed in Shapely 2.0. Use the `geoms` property to access the
constituent parts of a multi-part geometry.
POINT (1 1)
POINT (2 2)
POINT (3 3)
```

In Shapely 2.0, all those examples will raise an error.

How do I update my code? To access the geometry parts of a multi-part geometry, you can use the `.geoms` attribute, as the warning indicates.

The examples above can be updated to:

```
>>> for part in mp.geoms:
...     print(part)
POINT (1 1)
POINT (2 2)
POINT (3 3)
>>> print(mp.geoms[1])
POINT (2 2)
>>> len(mp.geoms)
3
>>> list(mp.geoms)
[<shapely.geometry.point.Point at 0x7f2e0912bf10>,
 <shapely.geometry.point.Point at 0x7f2e09fed820>,
 <shapely.geometry.point.Point at 0x7f2e09fed4c0>]
```

The single-part geometries (Point, LineString, Polygon) already didn't support those features, and for those classes there is no change in behaviour for this aspect.

5.3.3 Interoperability with NumPy and the array interface

Conversion of the coordinates to (NumPy) arrays

Shapely provides an array interface to have easy access to the coordinates as, for example, NumPy arrays (*manual section*).

A small example:

```
>>> line = LineString([(0, 0), (1, 1), (2, 2)])
>>> import numpy as np
>>> np.asarray(line)
array([[0., 0.],
       [1., 1.],
       [2., 2.]])
```

In addition, there are also the explicit `array_interface()` method and `ctypes` attribute to get access to the coordinates as array data:

```
>>> line.ctypes
<shapely.geometry.linestring.c_double_Array_6 at 0x7f75261eb740>
>>> line.array_interface()
{'version': 3,
 'typestr': '<f8',
 'data': <shapely.geometry.linestring.c_double_Array_6 at 0x7f752664ae40>,
 'shape': (3, 2)}
```

This functionality is available for Point, LineString, LinearRing and MultiPoint.

For more robust interoperability with NumPy, this array interface will be removed from those geometry classes, and limited to the `coords`.

Starting with Shapely 1.8, converting a geometry object to a NumPy array directly will start raising a warning:

```
>>> np.asarray(line)
ShapelyDeprecationWarning: The array interface is deprecated and will no longer
work in Shapely 2.0. Convert the '.coords' to a NumPy array instead.
array([[0., 0.],
       [1., 1.],
       [2., 2.]])
```

How do I update my code? To convert a geometry to a NumPy array, you can convert the `.coords` attribute instead:

```
>>> line.coords
<shapely.coords.CoordinateSequence at 0x7f2e09e88d60>
>>> np.array(line.coords)
array([[0., 0.],
       [1., 1.],
       [2., 2.]])
```

The `array_interface()` method and `ctypes` attribute will be removed in Shapely 2.0, but since Shapely will start requiring NumPy as a dependency, you can use NumPy or its array interface directly. Check the NumPy docs on the `ctypes` attribute or the array interface for more details.

Creating NumPy arrays of geometry objects

Shapely geometry objects can be stored in NumPy arrays using the `object` dtype. In general, one could create such an array from a list of geometries as follows:

```
>>> from shapely.geometry import Point
>>> arr = np.array([Point(0, 0), Point(1, 1), Point(2, 2)])
>>> arr
array([<shapely.geometry.point.Point object at 0x7fb798407cd0>,
       <shapely.geometry.point.Point object at 0x7fb7982831c0>,
       <shapely.geometry.point.Point object at 0x7fb798283b80>],
      dtype=object)
```

The above works for point geometries, but because in Shapely 1.x, some geometry types are sequence-like (see above), NumPy can try to "unpack" them when creating an array. Therefore, for more robust creation of a NumPy array from a list of geometries, it's generally recommended to this in a two-step way (first creating an empty array and then filling it):

```
geoms = [Point(0, 0), Point(1, 1), Point(2, 2)]
arr = np.empty(len(geoms), dtype="object")
arr[:] = geoms
```

This code snippet results in the same array as the example above, and works for all geometry types and Shapely/NumPy versions.

However, starting with Shapely 1.8, the above code will show deprecation warnings that cannot be avoided (depending on the geometry type, NumPy tries to access the array interface of the objects or check if an object is iterable or has a length, and those operations are all deprecated now. The end result is still correct, but the warnings appear nonetheless). Specifically in this case, it is fine to ignore those warnings (and the only way to make them go away):

```
import warnings
from shapely.errors import ShapelyDeprecationWarning

geoms = [Point(0, 0), Point(1, 1), Point(2, 2)]
arr = np.empty(len(geoms), dtype="object")

with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=ShapelyDeprecationWarning)
    arr[:] = geoms
```

In Shapely 2.0, the geometry objects will no longer be sequence like and those deprecation warnings will be removed (and thus the `filterwarnings` will no longer be necessary), and creation of NumPy arrays will generally be more robust.

If you maintain code that depends on Shapely, and you want to have it work with multiple versions of Shapely, the above code snippet provides a context manager that can be copied into your project:

```
import contextlib
import shapely
import warnings
from packaging import version  # https://packaging.pypa.io/

SHAPELY_GE_20 = version.parse(shapely.__version__) >= version.parse("2.0a1")

try:
```

(continues on next page)

(continued from previous page)

```
    from shapely.errors import ShapelyDeprecationWarning as shapely_warning
except ImportError:
    shapely_warning = None

if shapely_warning is not None and not SHAPELY_GE_20:
    @contextlib.contextmanager
    def ignore_shapely2_warnings():
        with warnings.catch_warnings():
            warnings.filterwarnings("ignore", category=shapely_warning)
            yield
else:
    @contextlib.contextmanager
    def ignore_shapely2_warnings():
        yield
```

This can then be used when creating NumPy arrays (be careful to *only* use it for this specific purpose, and not generally suppress those warnings):

```
geoms = [...]
arr = np.empty(len(geoms), dtype="object")
with ignore_shapely2_warnings():
    arr[:] = geoms
```

5.3.4 Consistent creation of empty geometries

Shapely 1.x is inconsistent in creating empty geometries between various creation methods. A small example for an empty Polygon geometry:

```
# Using an empty constructor results in a GeometryCollection
>>> from shapely.geometry import Polygon
>>> g1 = Polygon()
>>> type(g1)
<class 'shapely.geometry.polygon.Polygon'>
>>> g1.wkt
GEOMETRYCOLLECTION EMPTY

# Converting from WKT gives a correct empty polygon
>>> from shapely import wkt
>>> g2 = wkt.loads("POLYGON EMPTY")
>>> type(g2)
<class 'shapely.geometry.polygon.Polygon'>
>>> g2.wkt
POLYGON EMPTY
```

Shapely 1.8 does not yet change this inconsistent behaviour, but starting with Shapely 2.0, the different methods will always consistently give an empty geometry object of the correct type, instead of using an empty GeometryCollection as "generic" empty geometry object.

How do I update my code? Those cases that will change don't raise a warning, but you will need to update your code if you rely on the fact that empty geometry objects are of the GeometryCollection type. Use the `.is_empty` attribute for robustly checking if a geometry object is an empty geometry.

In addition, the WKB serialization methods will start supporting empty Points (using `"POINT (NaN NaN)"` to represent an empty point).

5.3.5 Other deprecated functionality

There are some other various functions and methods deprecated in Shapely 1.8 as well:

- The adapters to create geometry-like proxy objects with coordinates stored outside Shapely geometries are deprecated and will be removed in Shapely 2.0 (e.g. created using `asShape()`). They have little to no benefit compared to the normal geometry classes, as thus you can convert to your data to a normal geometry object instead. Use the `shape()` function instead to convert a GeoJSON-like dict to a Shapely geometry.
- The `empty()` method on a geometry object is deprecated.
- The `shapely.ops.cascaded_union` function is deprecated. Use `shapely.ops.unary_union` instead, which internally already uses a cascaded union operation for better performance.

5.4 Migrating from PyGEOS

The PyGEOS package was merged with Shapely in December 2021 and will be released as part of Shapely 2.0. No further development will take place for the PyGEOS package (except for providing up to date packages until Shapely 2.0 is released).

Therefore, everybody using PyGEOS is highly recommended to migrate to Shapely 2.0.

Generally speaking, this should be a smooth experience because all functionality of PyGEOS was added to Shapely. All vectorized functions availabe in `pygeos` have been added to the top-level `shapely` module, with only minor differences (see below). Migrating from PyGEOS to Shapely 2.0 can thus be done by replacing the `pygeos` import and module calls:

```
import pygeos
polygon = pygeos.box(0, 0, 2, 2)
points = pygeos.points(...)
pygeos.contains(polygon, points)
```

Using Shapely 2.0, this can now be written as:

```
import shapely
polygon = shapely.box(0, 0, 2, 2)
points = shapely.points(...)
shapely.contains(polygon, points)
```

In addition, you now also have the scalar interface of Shapely which wasn't implemented in PyGEOS.

5.4.1 Differences between PyGEOS and Shapely 2.0

STRtree API changes

Functionality-wise, everything from `pygeos.STRtree` is available in Shapely 2.0. But while merging into Shapely, some methods have been changed or merged:

- The `query()` and `query_bulk()` methods have been merged into a single `query()` method. The `query()` method now accepts an array of geometries as well in addition to a single geometry, and in that case it will return 2D array of indices.

  It should thus be a matter of replacing `query_bulk` with `query` in your code.

  See *STRtree.query()* for more details.

- The `nearest()` method was changed to return an array of the same shape as the input geometries. Thus, for a scalar geometry it now returns a single integer index (instead of a (2, 1) array), and for an array of geometries it now returns a 1D array of indices ((n,) array instead of a (2, n) array).

  See *STRtree.nearest()* for more details.

- The `nearest_all()` method has been replaced with `query_nearest()`. For an array of geometries, the output is the same, but when passing a scalar geometry as input, the method now returns a 1D array instead of a 2D array (consistent with `query()`).

  In addition, this method gained the new `exclusive` and `all_matches` keywords (with defaults preserving existing behaviour from PyGEOS). See *STRtree.query_nearest()* for more details.

Other differences

- The `pygeos.Geometry(..)` constructor has not been retained in Shapely (the class exists as base class, but the constructor is not callable). Use one of the subclasses, or `shapely.from_wkt(..)`, instead.
- The `apply()` function was renamed to `transform()`.
- The `tolerance` keyword of the `segmentize()` function was renamed to `max_segment_length`.
- The `quadsegs` keyword of the `buffer()` and `offset_curve()` functions was renamed to `quad_segs`.
- The `preserve_topology` keyword of `simplify()` now defaults to `True` instead of `False`.
- The behaviour of `union_all()` / `intersection_all()` / `symmetric_difference_all` was changed to return an empty GeometryCollection for an empty or all-None sequence as input (instead of returning None).
- The `radius` keyword of the `buffer()` funtion was renamed to `distance`.

5.5 Release notes

5.5.1 Version 2.x

Version 2.0.4 (2024-04-16)

Bug fixes:

- Fix bug in `to_wkt` with multiple empty Z geometries (#2012).
- Fix bug in `to_ragged_array` for an array of Points with missing values (#2034).

Wheels for Python versions >= 3.9 will be compatible with the upcoming NumPy 2.0 release (as well as with supported NumPy 1.x versions).

Version 2.0.3 (2024-02-16)

Bug fixes:

- Fix regression in the `oriented_envelope` ufunc to accept array-like input in case of GEOS<3.12 (#1929).

Packaging related:

- The binary wheels are not yet compatible with a future NumPy 2.0 release, therefore a `numpy<2` upper pin was added to the requirements (#1972).
- Upgraded the GEOS version in the binary wheel distributions to 3.11.3.

Version 2.0.2 (2023-10-12)

Bug fixes:

- Fix regression in the (in)equality comparison (`geom1 == geom2`) using `__eq__` to not ignore the z-coordinates (#1732).
- Fix `MultiPolygon()` constructor to accept polygons without holes (#1850).
- Fix `minimum_rotated_rectangle()` (`oriented_envelope()`) to always return the minimum area solution (instead of minimum width). In practice, it will use the GEOS implementation only for GEOS 3.12+, and for older GEOS versions fall back to the implementation that was included in Shapely < 2.

Wheels are available for Python 3.12 (and still include GEOS 3.11.2). Building from source is now compatible with Cython 3.

Version 2.0.1 (2023-01-30)

Bug fixes:

- Fix regression in the `Polygon()` constructor taking a sequence of Points (#1662).
- Fix regression in the geometry constructors when passing `decimal.Decimal` coordinate values (#1707).
- Fix `STRtree()` to not make the passed geometry array immutable as side-effect of the constructor (#1714).
- Fix the `directed` keyword in `shapely.ops.linemerge()` (#1695).

Improvements:

- Expose the function to get a matplotlib Patch object from a (Multi)Polygon (without already plotting it) publicly as `shapely.plotting.patch_from_polygon()` (#1704).

Acknowledgments

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

- Brendan Ward
- Erik Pettersson +
- Hood Chatham +
- Idan Miara +
- Joris Van den Bossche
- Martin Fleischmann
- Michał Górny +
- Sebastian Castro +

Version 2.0.0 (2022-12-12)

Shapely 2.0 version is a major release featuring a complete refactor of the internals and new vectorized (element-wise) array operations, providing considerable performance improvements (based on the developments in the PyGEOS package), along with several breaking API changes and many feature improvements.

For more background, see RFC 1: Roadmap for Shapely 2.0.

Refactor of the internals

Shapely wraps the GEOS C++ library for use in Python. Before 2.0, Shapely used `ctypes` to link to GEOS at runtime, but doing so resulted in extra overhead and installation challenges. With 2.0, the internals of Shapely have been refactored to expose GEOS functionality through a Python C extension module that is compiled in advance.

The pointer to the actual GEOS Geometry object is stored in a lightweight Python extension type. A single *Geometry* Python extension type is defined in C wrapping a *GEOSGeometry* pointer. This extension type is further subclassed in Python to provide the geometry type-specific classes from Shapely (Point, LineString, Polygon, etc). The GEOS pointer is accessible from C as a static attribute of the Python object (an attribute of the C struct that makes up a Python object), which enables using vectorized functions within C and thus avoiding Python overhead while looping over an array of geometries (see next section).

Vectorized (element-wise) geometry operations

Before the 2.0 release, Shapely only provided an interface for scalar (individual) geometry objects. Users had to loop over individual geometries within an array of geometries and call scalar methods or properties, which is both more verbose to use and has a large performance overhead.

Shapely 2.0 exposes GEOS operations as vectorized functions that operate on arrays of geometries using a familiar NumPy interface. Those functions are implemented as NumPy universal functions (or ufunc for short). A universal function is a function that operates on n-dimensional arrays in an element-by-element fashion and supports array broadcasting. All loops over geometries are implemented in C, which results in substantial performance improvements when performing operations using many geometries. This also allows operations to be less verbose.

NumPy is now a required dependency.

An example of this functionality using a small array of points and a single polygon:

```
>>> import shapely
>>> from shapely import Point, box
>>> import numpy as np
>>> geoms = np.array([Point(0, 0), Point(1, 1), Point(2, 2)])
>>> polygon = box(0, 0, 2, 2)
```

Before Shapely 2.0, a `for` loop was required to operate over an array of geometries:

```
>>> [polygon.contains(point) for point in geoms]
[False,  True, False]
```

In Shapely 2.0, we can now compute whether the points are contained in the polygon directly with one function call:

```
>>> shapely.contains(polygon, geoms)
array([False,  True, False])
```

This results in a considerable speedup, especially for larger arrays of geometries, as well as a nicer user interface that avoids the need to write `for` loops. Depending on the operation, this can give a performance increase with factors of 4x to 100x. In general, the greatest speedups are for lightweight GEOS operations, such as `contains`, which would previously have been dominated by the high overhead of `for` loops in Python. See https://caspervdw.github.io/Introducing-Pygeos/ for more detailed examples.

The new vectorized functions are available in the top-level `shapely` namespace. All the familiar geospatial methods and attributes from the geometry classes now have an equivalent as top-level function (with some small name deviations, such as the `.wkt` attribute being available as a `to_wkt()` function). Some methods from submodules (for example, several functions from the `shapely.ops` submodule such as `polygonize()`) are also made available in a vectorized version as top-level function.

A full list of functions can be found in the API docs (see the pages listed under "API REFERENCE" in the left sidebar).

- Vectorized constructor functions
- Optionally output to a user-specified array (`out` keyword argument) when constructing geometries from `indices`.
- Enable bulk construction of geometries with different number of coordinates by optionally taking index arrays in all creation functions.

Shapely 2.0 API changes (deprecated in 1.8)

The Shapely 1.8 release included several deprecation warnings about API changes that would happen in Shapely 2.0 and that can be fixed in your code (making it compatible with both <=1.8 and >=2.0). See *Migrating to Shapely 1.8 / 2.0* for more details on how to update your code.

It is highly recommended to first upgrade to Shapely 1.8 and resolve all deprecation warnings before upgrading to Shapely 2.0.

Summary of changes:

- Geometries are now immutable and hashable.
- Multi-part geometries such as MultiPolygon no longer behave as "sequences". This means that they no longer have a length, are not iterable, and are not indexable anymore. Use the `.geoms` attribute instead to access individual parts of a multi-part geometry.
- Geometry objects no longer directly implement the numpy array interface to expose their coordinates. To convert to an array of coordinates, use the `.coords` attribute instead (`np.asarray(geom.coords)`).

- The following attributes and methods on the Geometry classes were previously deprecated and are now removed from Shapely 2.0:
  - `array_interface()` and `ctypes`
  - `asShape()`, and the adapters classes to create geometry-like proxy objects (use `shape()` instead).
  - `empty()` method

Some new deprecations have been introduced in Shapely 2.0:

- Directly calling the base class `BaseGeometry()` constructor or the `EmptyGeometry()` constructor is deprecated and will raise an error in the future. To create an empty geometry, use one of the subclasses instead, for example `GeometryCollection()` (#1022).
- The `shapely.speedups` module (the `enable` and `disable` functions) is deprecated and will be removed in the future. The module no longer has any affect in Shapely >=2.0.

Breaking API changes

Some additional backwards incompatible API changes were included in Shapely 2.0 that were not deprecated in Shapely 1.8:

- Consistent creation of empty geometries (for example `Polygon()` now actually creates an empty Polygon instead of an empty geometry collection).
- The `.bounds` attribute of an empty geometry now returns a tuple of NaNs instead of an empty tuple (#1023).
- The `preserve_topology` keyword of `simplify()` now defaults to `True` (#1392).
- A `GeometryCollection` that consists of all empty sub-geometries now returns those empty geometries from its `.geoms` attribute instead of returning an empty list (#1420).
- The `Point(..)` constructor no longer accepts a sequence of coordinates consisting of more than one coordinate pair (previously, subsequent coordinates were ignored) (#1600).
- The unused `shape_factory()` method and `HeterogeneousGeometrySequence` class are removed (#1421).
- The undocumented `__geom__` attribute has been removed. If necessary (although not recommended for use beyond experimentation), use the `_geom` attribute to access the raw GEOS pointer (#1417).
- The `logging` functionality has been removed. All error messages from GEOS are now raised as Python exceptions (#998).
- Several custom exception classes defined in `shapely.errors` that are no longer used internally have been removed. Errors from GEOS are now raised as `GEOSException` (#1306).

The `STRtree` interface has been substantially changed. See the section *below* for more details.

Additionally, starting with GEOS 3.11 (which is included in the binary wheels on PyPI), the behaviour of the `parallel_offset` (`offset_curve`) method changed regarding the orientation of the resulting line. With GEOS < 3.11, the line retains the same direction for a left offset (positive distance) or has opposite direction for a right offset (negative distance), and this behaviour was documented as such in previous Shapely versions. Starting with GEOS 3.11, the function tries to preserve the orientation of the original line.

New features

Geometry subclasses are now available in the top-level namespace

Following the new vectorized functions in the top-level `shapely` namespace, the Geometry subclasses (`Point`, `LineString`, `Polygon`, etc) are now available in the top-level namespace as well. Thus it is no longer needed to import those from the `shapely.geometry` submodule.

The following:

```
from shapely.geometry import Point
```

can be replaced with:

```
from shapely import Point
```

or:

```
import shapely
shapely.Point(...)
```

Note: for backwards compatibility (and being able to write code that works for both <=1.8 and >2.0), those classes still remain accessible from the `shapely.geometry` submodule as well.

More informative repr with truncated WKT

The repr (`__repr__`) of Geometry objects has been simplified and improved to include a descriptive Well-Known-Text (WKT) formatting. Instead of showing the class name and id:

```
>>> Point(0, 0)
<shapely.geometry.point.Point at 0x7f0b711f1310>
```

we now get:

```
>>> Point(0, 0)
<POINT (0 0)>
```

For large geometries with many coordinates, the output gets truncated to 80 characters.

Support for fixed precision model for geometries and in overlay functions

GEOS 3.9.0 overhauled the overlay operations (union, intersection, (symmetric) difference). A complete rewrite, dubbed "OverlayNG", provides a more robust implementation (no more TopologyExceptions even on valid input), the ability to specify the output precision model, and significant performance optimizations. When installing Shapely with GEOS >= 3.9 (which is the case for PyPI wheels and conda-forge packages), you automatically get these improvements (also for previous versions of Shapely) when using the overlay operations.

Shapely 2.0 also includes the ability to specify the precision model directly:

- The *set_precision()* function can be used to conform a geometry to a certain grid size (may round and reduce coordinates), and this will then also be used by subsequent overlay methods. A *get_precision()* function is also available to inspect the precision model of geometries.
- The `grid_size` keyword in the overlay methods can also be used to specify the precision model of the output geometry (without first conforming the input geometries).

Releasing the GIL for multithreaded applications

Shapely itself is not multithreaded, but its functions generally allow for multithreading by releasing the Global Interpreter Lock (GIL) during execution. Normally in Python, the GIL prevents multiple threads from computing at the same time. Shapely functions internally release this constraint so that the heavy lifting done by GEOS can be done in parallel, from a single Python process.

STRtree API changes and improvements

The biggest change in the *STRtree* interface is that all operations now return indices of the input tree or query geometries, instead of the geometries itself. These indices can be used to index into anything associated with the input geometries, including the input geometries themselves, or custom items stored in another object of the same length and order as the geometries.

In addition, Shapely 2.0 includes several improvements to `STRtree`:

- Directly include predicate evaluation in *STRtree.query()* by specifying the `predicate` keyword. If a predicate is provided, tree geometries with bounding boxes that overlap the bounding boxes of the input geometries are further filtered to those that meet the predicate (using prepared geometries under the hood for efficiency).
- Query multiple input geometries (spatial join style) with *STRtree.query()* by passing an array of geometries. In this case, the return value is a 2D array with shape (2, n) where the subarrays correspond to the indices of the input geometries and indices of the tree geometries associated with each.
- A new *STRtree.query_nearest()* method was added, returning the index of the nearest geometries in the tree for each input geometry. Compared to *STRtree.nearest()*, which only returns the index of a single nearest geometry for each input geometry, this new methods allows for:
  - returning all equidistant nearest geometries,
  - excluding nearest geometries that are equal to the input,
  - specifying an `max_distance` to limit the search radius, potentially increasing the performance,
  - optionally returning the distance.
- Fixed `STRtree` creation to allow querying the tree in a multi-threaded context.

Bindings for new GEOS functionalities

Several (new) functions from GEOS are now exposed in Shapely:

- *hausdorff_distance()* and *frechet_distance()*
- *contains_properly()*
- *extract_unique_points()*
- *reverse()*
- *node()*
- *contains_xy()* and *intersects_xy()*
- *build_area()* (GEOS >= 3.8)
- *minimum_bounding_circle()* and *minimum_bounding_radius()* (GEOS >= 3.8)
- *coverage_union()* and *coverage_union_all()* (GEOS >= 3.8)
- *segmentize()* (GEOS >= 3.10)

- *dwithin()* (GEOS >= 3.10)
- *remove_repeated_points()* (GEOS >= 3.11)
- *line_merge()* added *directed* parameter (GEOS > 3.11)
- *concave_hull()* (GEOS >= 3.11)

In addition some aliases for existing methods have been added to provide a method name consistent with GEOS or PostGIS:

- *line_interpolate_point()* (`interpolate`)
- *line_locate_point()* (`project`)
- *offset_curve()* (`parallel_offset`)
- *point_on_surface()* (`representative_point`)
- *oriented_envelope()* (`minimum_rotated_rectangle`)
- *delaunay_triangles()* (`ops.triangulate`)
- *voronoi_polygons()* (`ops.voronoi_diagram`)
- *shortest_line()* (`ops.nearest_points`)
- *is_valid_reason()* (`validation.explain_validity`)

Getting information / parts / coordinates from geometries

A set of GEOS getter functions are now also exposed to inspect geometries:

- *get_dimensions()*
- *get_coordinate_dimension()*
- *get_srid()*
- *get_num_points()*
- *get_num_interior_rings()*
- *get_num_geometries()*
- *get_num_coordinates()*
- *get_precision()*

Several functions are added to extract parts:

- *get_geometry()* to get a geometry from a GeometryCollection or Multi-part geometry.
- *get_exterior_ring()* and *get_interior_ring()* to get one of the rings of a Polygon.
- *get_point()* to get a point (vertex) of a linestring or linearring.
- *get_x()*, *get_y()* and *get_z()* to get the x/y/z coordinate of a Point.

Methods to extract all parts or coordinates at once have been added:

- The *get_parts()* function can be used to get individual parts of an array of multi-part geometries.
- The *get_rings()* function, similar as `get_parts` but specifically to extract the rings of Polygon geometries.
- The *get_coordinates()* function to get all coordinates from a geometry or array of goemetries as an array of floats.

Each of those three functions has an optional `return_index` keyword, which allows to also return the indexes of the original geometries in the source array.

Prepared geometries

Prepared geometries are now no longer separate objects, but geometry objects themselves can be prepared (this makes the `shapely.prepared` module superfluous).

The *prepare()* function generates a GEOS prepared geometry which is stored on the Geometry object itself. All binary predicates (except `equals`) will make use of this if the input geometry has already been prepared. Helper functions *destroy_prepared()* and *is_prepared()* are also available.

New IO methods (GeoJSON, ragged arrays)

- Added GeoJSON input/output capabilities *from_geojson()* and *to_geojson()* for GEOS >= 3.10.
- Added conversion to/from ragged array representation using a contiguous array of coordinates and offset arrays: *to_ragged_array()* and *from_ragged_array()*.

Other improvements

- Added *force_2d()* and *force_3d()* to change the dimensionality of the coordinates in a geometry.
- Addition of a *total_bounds()* function to return the outer bounds of an array of geometries.
- Added *empty()* to create a geometry array pre-filled with None or with empty geometries.
- Performance improvement in constructing LineStrings or LinearRings from numpy arrays for GEOS >= 3.10.
- Updated the *box()* ufunc to use internal C function for creating polygon (about 2x faster) and added `ccw` parameter to create polygon in counterclockwise (default) or clockwise direction.
- Start of a benchmarking suite using ASV.
- Added `shapely.testing.assert_geometries_equal`.

Bug fixes

- Fixed several corner cases in WKT and WKB serialization for varying GEOS versions, including:
  - Fixed the WKT serialization of single part 3D empty geometries to correctly include "Z" (for GEOS >= 3.9.0).
  - Handle empty points in WKB serialization by conversion to `POINT (nan, nan)` consistently for all GEOS versions (GEOS started doing this for >= 3.9.0).

Acknowledgments

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

- Adam J. Stewart +
- Alan D. Snow +
- Ariel Kadouri
- Bas Couwenberg
- Ben Beasley
- Brendan Ward +
- Casper van der Wel +
- Ewout ter Hoeven +
- Geir Arne Hjelle +
- James Gaboardi
- James Myatt +
- Joris Van den Bossche
- Keith Jenkins +
- Kian Meng Ang +
- Krishna Chaitanya +
- Kyle Barron
- Martin Fleischmann +
- Martin Lackner +
- Mike Taves
- Phil Chiu +
- Tanguy Ophoff +
- Tom Clancy
- Sean Gillies
- Giorgos Papadokostakis +
- Mattijn van Hoek +
- enrico ferreguti +
- gpapadok +
- mattijn +
- odidev +

5.5.2 Version 1.x

1.8.4 (2022-08-17)

Bug fixes:

- The new c_geom_p type caused a regression and has been removed (#1487).

1.8.3 (2022-08-16)

Deprecations:

The STRtree class will be changed in 2.0.0 and will not be compatible with the class in versions 1.8.x. This change obsoletes the deprecation announcement in 1.8a3 (below).

Packaging:

Wheels for 1.8.3 published on PyPI include GEOS 3.10.3.

Bug fixes:

- The signature for GEOSMinimumClearance has been corrected, fixing an issue affecting aarch64-darwin (#1480)
- Return and arg types have been corrected and made more strict for area, length, and distance properties.
- A new c_geom_p type has been created to replace c_void_p when calling GEOS functions (#1479).
- An incorrect polygon-line intersection (#1427) has been fixed in GEOS 3.10.3, which will be included in wheels published to PyPI.
- GEOS buffer parameters are now destroyed, fixing a memory leak (#1440).

1.8.2 (2022-05-03)

- Make Polygons and MultiPolygons closed by definition, like LinearRings. Resolves #1246.
- Perform frozen app check for GEOS before conda env check on macos as we already do on linux (#1301).
- Fix leak of GEOS coordinate sequence in nearest_points reported in #1098.

1.8.1.post1 (2022-02-17)

This post-release addresses a defect in the 1.8.1 source distribution. No .c files are included in the 1.8.1.post1 sdist and Cython is required to build and install from source.

1.8.1 (2022-02-16)

Packaging:

Wheels for 1.8.1 published on PyPI include GEOS 3.10.2. This version is the best version of GEOS yet. Discrepancies in behavior compared to previous versions are considered to be improvements.

For the first time, we will publish wheels for macos_arm64 (see PR #1310).

Python version support:

Shapely 1.8.1 works with Pythons 3.6-3.10.

Bug fixes:

- Require Cython >= 0.29.24 to support Python 3.10 (#1224).
- Fix array_interface_base (#1235).

1.8.0 (2021-10-25)

This is the final 1.8.0 release. There have been no changes since 1.8rc2.

1.8rc2 (2021-10-19)

Build:

A pyproject.toml file has been added to specify build dependencies for the _vectorized and _speedups modules (#1128). To install shapely without these build dependencies, use the features of your build tool that disable PEP 517 and 518 support.

Bug fixes:

- Part of PR #1042, which added a new primary GEOS library name to be searched for, has been reverted by PR #1201.

1.8rc1 (2021-10-04)

Deprecations:

The almost_exact() method of BaseGeometry has been deprecated. It is confusing and will be removed in 2.0.0. The equals_exact() method is to be used instead.

Bug fixes:

- We ensure that the _speedups module is always imported before _vectorized to avoid an unexplained condition on Windows with Python 3.8 and 3.9 (#1184).

1.8a3 (2021-08-24)

Deprecations:

The STRtree class deprecation warnings have been removed. The class in 2.0.0 will be backwards compatible with the class in 1.8.0.

Bug fixes:

- The __array_interface__ raises only AttributeError, all other exceptions are deprecated starting with Numpy 1.21 (#1173).
- The STRtree class now uses a pair of item, geom sequences internally instead of a dict (#1177).

1.8a2 (2021-07-15)

Python version support:

Shapely 1.8 will support only Python versions >= 3.6.

New features:

- The STRtree nearest*() methods now take an optional argument that specifies exclusion of the input geometry from results (#1115).
- A GeometryTypeError has been added to shapely.errors and is consistently raised instead of TypeError or ValueError as in version 1.7. For backwards compatibility, the new exception will derive from TypeError and Value error until version 2.0 (#1099).
- The STRtree class constructor now takes an optional second argument, a sequence of objects to be stored in the tree. If not provided, the sequence indices of the geometries will be stored, as before (#1112).
- The STRtree class has new query_geoms(), query_items(), nearest_geom(), and nearest_item() methods (#1112). The query() and nearest() methods remain as aliases for query_geoms() and nearest_geom().

Bug fixes:

- We no longer attempt to load libc to get the free function on Linux, but get it from the global symbol table.
- GEOS error messages printed when GEOS_getCoordSeq() is passed an empty geometry are avoided by never passing an empty geometry (#1134).
- Python's builtin super() is now used only as described in PEP 3135 (#1109).
- Only load conda GEOS dll if it exists (on Windows) (#1108).
- Add /opt/homebrew/lib to the list of directories to be searched for the GEOS shared library.
- Added new library search path to assist app creation with cx_Freeze.

1.8a1 (2021-03-03)

Shapely 1.8.0 will be a transitional version. There are a few bug fixes and new features, but it is mainly about warning of the upcoming changes in 2.0.0. Several more pre-releases before 1.8.0 are expected. See the migration guide to Shapely 1.8 / 2.0 for more details on how to update your code (https://shapely.readthedocs.io/en/latest/migration.html).

Python version support:

Shapely 1.8 will support only Python versions >= 3.5 (#884).

Deprecations:

The following functions and geometry attributes and methods will be removed in version 2.0.0.

- ops.cascaded_union
- geometry .empty()
- geometry .ctypes and .__array_interface__
- multi-part geometry .__len__
- setting custom attributes on geometry objects

Geometry objects will become immutable in version 2.0.0.

The STRtree class will be entirely changed in 2.0.0. The exact future API is not yet decided, but will be decided before 1.8.0 is released.

Deprecation warnings will be emitted in 1.8a1 when any of these features are used.

The deprecated .to_wkb() and .to_wkt() methods on the geometry objects have been removed.

New features:

- Add a normalize() method to geometry classes, exposing the GEOSNormalize algorithm (#1090).
- Initialize STRtree with a capacity of 10 items per node (#1070).
- Load libraries relocated to shapely/.libs by auditwheel versions < 3.1 or relocated to Shapely.libs by auditwheel versions >= 3.1.
- shapely.ops.voronoi_diagram() computes the Voronoi Diagram of a geometry or geometry collection (#833, #851).
- shapely.validation.make_valid() fixes invalid geometries (#883)

Bug fixes:

- For pyinstaller we now handle the case of more than one GEOS library in the environment, such as when fiona and rasterio wheels are co-installed with shapely (#1071).
- The ops.split function now splits on touch to eliminate confusing discrepancies between results using multi and single part splitters (#1034).
- Several issues with duplication and order of vertices in ops.substring have been fixed (#1008).

Packaging:

- The wheels uploaded to PyPI will include GEOS 3.9.1.

1.7.1 (2020-08-20)

- `STRtree` now safely implements the pickle protocol (#915).
- Documentation has been added for `minimum_clearance` (#875, #874).
- In `STRtree.__del__()` we guard against calling `GEOSSTRtree_destroy` when the lgeos module has already been torn down on exit (#897, #830).
- Documentation for the `overlaps()` method has been corrected (#920).
- Correct the test in `shapely.geometry.base.BaseGeometry.empty()` to eliminate memory leaks like the one reported in #745.
- Get free() not from libc but from the processes global symbols (#891), fixing a bug that manifests on OS X 10.15 and 10.16.
- Extracting substrings from complex lines has been made more correct (#848, #849).
- Splitting of complex geometries has been sped up by preparing the input geometry (#871).
- Fix bug in concatenation of function argtypes (#866).
- Improved documentation of STRtree usage (#857).
- Improved handling for empty list or list of lists in GeoJSON coordinates (#852).
- The polylabel algorithm now accounts for polygon holes (#851, #817).

1.7.0 (2020-01-28)

This is the final 1.7.0 release. There have been no changes since 1.7b1.

1.7b1 (2020-01-13)

First beta release.

1.7a3 (2019-12-31)

New features:

- The buffer operation can now be single-sides (#806, #727).

Bug fixes:

- Add /usr/local/lib to the list of directories to be searched for the GEOS shared library (#795).
- ops.substring now returns a line with coords in end-to-front order when given a start position that is greater than the end position (#628).
- Implement `__bool__()` for geometry base classes so that `bool(geom)` returns the logical complement of `geom.is_empty` (#754).
- Remove assertion on the number of version-like strings found in the GEOS version string. It could be 2 or 3.

1.7a2 (2019-06-21)

- Nearest neighbor search has been added to STRtree (#668).
- Disallow sequences of MultiPolygons as arguments to the MultiPolygon constructor, resolving #588.
- Removed vendorized *functools* functions previously used to support Python 2.5.

Bug fixes:

- Avoid reloading the GEOS shared library when using an installed binary wheel on OS X (#735), resolving issue #553.
- The shapely.ops.orient function can now orient multi polygons and geometry collections as well as polygons (#733).
- Polygons can now be constructed from sequences of point objects as well as sequences of x, y sequences (#732).
- The exterior of an empty polygon is now equal to an empty linear ring (#731).
- The bounds property of an empty point object now returns an empty tuple, consistent with other geometry types (#723).
- Segmentation faults when non-string values are passed to the WKT loader are avoided by #700.
- Failure of ops.substring when the sub linestring coincides with the beginning of the linestring has been fixed (#658).
- Segmentation faults from interpolating on an empty linestring are prevented by #655.
- A missing special case for rectangular polygons has been added to the polylabel algorithm (#644).
- LinearRing can be created from a LineString (#638).
- The prepared geometry validation condition has been tightened in #632 to fix the bug reported in #631.

- Attempting to interpolate an empty geometry no longer results in a segmentation fault, raising *ValueError* instead (#653).

1.7a1 (2018-07-29)

New features:

- A Python version check is made by the package setup script. Shapely 1.7 supports only Python versions 2.7 and 3.4+ (#610).
- Added a new *EmptyGeometry* class to support GeoPandas (#514).
- Added new *shapely.ops.substring* function (#459).
- Added new *shapely.ops.clip_by_rect* function (#583).
- Use DLLs indicated in sys._MEIPASS' to support PyInstaller frozen apps (#523).
- *shapely.wkb.dumps* now accepts an *srid* integer keyword argument to write WKB data including a spatial reference ID in the output data (#593).

Bug fixes:

- *shapely.geometry.shape* can now marshal empty GeoJSON representations (#573).
- An exception is raised when an attempt is made to *prepare* a *PreparedGeometry* (#577, #595).
- Keyword arguments have been removed from a geometry object's *wkt* property getter (#581, #594).

1.6.4.post1 (2018-01-24)

- Fix broken markup in this change log, which restores our nicely formatted readme on PyPI.

1.6.4 (2018-01-24)

- Handle a `TypeError` that can occur when geometries are torn down (#473, #528).

1.6.3 (2017-12-09)

- AttributeError is no longer raised when accessing __geo_interface__ of an empty polygon (#450).
- `asShape` now handles empty coordinates in mappings as `shape` does (#542). Please note that `asShape` is likely to be deprecated in a future version of Shapely.
- Check for length of LineString coordinates in speed mode, preventing crashes when using LineStrings with only one coordinate (#546).

1.6.2 (2017-10-30)

- A 1.6.2.post1 release has been made to fix a problem with macosx wheels uploaded to PyPI.

1.6.2 (2017-10-26)

- Splitting a linestring by one of its end points will now succeed instead of failing with a `ValueError` (#524, #533).
- Missing documentation of a geometry's `overlaps` predicate has been added (#522).

1.6.1 (2017-09-01)

- Avoid `STRTree` crashes due to dangling references (#505) by maintaining references to added geometries.
- Reduce log level to debug when reporting on calls to ctypes `CDLL()` that don't succeed and are retried (#515).
- Clarification: applications like GeoPandas that need an empty geometry object should use `BaseGeometry()` instead of `Point()` or `Polygon()`. An `EmptyGeometry` class has been added in the master development branch and will be available in the next non-bugfix release.

1.6.0 (2017-08-21)

Shapely 1.6.0 adds new attributes to existing geometry classes and new functions (`split()` and `polylabel()`) to the shapely.ops module. Exceptions are consolidated in a shapely.errors module and logging practices have been improved. Shapely's optional features depending on Numpy are now gathered into a requirements set named "vectorized" and these may be installed like `pip install shapely[vectorized]`.

Much of the work on 1.6.0 was aimed to improve the project's build and packaging scripts and to minimize run-time dependencies. Shapely now vendorizes packaging to use during builds only and never again invokes the geos-config utility at run-time.

In addition to the changes listed under the alpha and beta pre-releases below, the following change has been made to the project:

- Project documentation is now hosted at https://shapely.readthedocs.io/en/latest/.

Thank you all for using, promoting, and contributing to the Shapely project.

1.6b5 (2017-08-18)

Bug fixes:

- Passing a single coordinate to `LineString()` with speedups disabled now raises a ValueError as happens with speedups enabled. This resolves #509.

1.6b4 (2017-02-15)

Bug fixes:

- Isolate vendorized packaging in a _vendor directory, remove obsolete dist-info, and remove packaging from project requirements (resolves #468).

1.6b3 (2016-12-31)

Bug fixes:

- Level for log messages originating from the GEOS notice handler reduced from WARNING to INFO (#447).
- Permit speedups to be imported again without Numpy (#444).

1.6b2 (2016-12-12)

New features:

- Add support for GeometryCollection to shape and asShape functions (#422).

1.6b1 (2016-12-12)

Bug fixes:

- Implemented __array_interface__ for empty Points and LineStrings (#403).

1.6a3 (2016-12-01)

Bug fixes:

- Remove accidental hard requirement of Numpy (#431).

Packaging:

- Put Numpy in an optional requirement set named "vectorized" (#431).

1.6a2 (2016-11-09)

Bug fixes:

- Shapely no longer configures logging in `geos.py` (#415).

Refactoring:

- Consolidation of exceptions in `shapely.errors`.
- `UnsupportedGEOSVersionError` is raised when GEOS < 3.3.0 (#407).

Packaging:

- Added new library search paths to assist Anaconda (#413).
- geos-config will now be bypassed when NO_GEOS_CONFIG env var is set. This allows configuration of Shapely builds on Linux systems that for whatever reasons do not include the geos-config program (#322).

1.6a1 (2016-09-14)

New features:

- A new error derived from NotImplementedError, with a more useful message, is raised when the GEOS backend doesn't support a called method (#216).
- The `project()` method of LineString has been extended to LinearRing geometries (#286).
- A new `minimum_rotated_rectangle` attribute has been added to the base geometry class (#354).
- A new `shapely.ops.polylabel()` function has been added. It computes a point suited for labeling concave polygons (#395).
- A new `shapely.ops.split()` function has been added. It splits a geometry by another geometry of lesser dimension: polygon by line, line by point (#293, #371).
- `Polygon.from_bounds()` constructs a Polygon from bounding coordinates (#392).
- Support for testing with Numpy 1.4.1 has been added (#301).
- Support creating all kinds of empty geometries from empty lists of Python objects (#397, #404).

Refactoring:

- Switch from `SingleSidedBuffer()` to `OffsetCurve()` for GEOS >= 3.3 (#270).
- Cython speedups are now enabled by default (#252).

Packaging:

- Packaging 16.7, a setup dependency, is vendorized (#314).
- Infrastructure for building manylinux1 wheels has been added (#391).
- The system's `geos-config` program is now only checked when `setup.py` is executed, never during normal use of the module (#244).
- Added new library search paths to assist PyInstaller (#382) and Windows (#343).

1.5.17 (2016-08-31)

- Bug fix: eliminate memory leak in geom_factory() (#408).
- Bug fix: remove mention of negative distances in parallel_offset and note that vertices of right hand offset lines are reversed (#284).

1.5.16 (2016-05-26)

- Bug fix: eliminate memory leak when unpickling geometry objects (#384, #385).
- Bug fix: prevent crashes when attempting to pickle a prepared geometry, raising `PicklingError` instead (#386).
- Packaging: extension modules in the OS X wheels uploaded to PyPI link only libgeos_c.dylib now (you can verify and compare to previous releases with `otool -L shapely/vectorized/_vectorized.so`).

1.5.15 (2016-03-29)

- Bug fix: use uintptr_t to store pointers instead of long in _geos.pxi, preventing an overflow error (#372, #373). Note that this bug fix was erroneously reported to have been made in 1.5.14, but was not.

1.5.14 (2016-03-27)

- Bug fix: use `type()` instead of `isinstance()` when evaluating geometry equality, preventing instances of base and derived classes from being mistaken for equals (#317).
- Bug fix: ensure that empty geometries are created when constructors have no args (#332, #333).
- Bug fix: support app "freezing" better on Windows by not relying on the `__file__` attribute (#342, #377).
- Bug fix: ensure that empty polygons evaluate to be == (#355).
- Bug fix: filter out empty geometries that can cause segfaults when creating and loading STRtrees (#345, #348).
- Bug fix: no longer attempt to reuse GEOS DLLs already loaded by Rasterio or Fiona on OS X (#374, #375).

1.5.13 (2015-10-09)

- Restore setup and runtime discovery and loading of GEOS shared library to state at version 1.5.9 (#326).
- On OS X we try to reuse any GEOS shared library that may have been loaded via import of Fiona or Rasterio in order to avoid a bug involving the GEOS AbstractSTRtree (#324, #327).

1.5.12 (2015-08-27)

- Remove configuration of root logger from libgeos.py (#312).
- Skip test_fallbacks on Windows (#308).
- Call setlocale(locale.LC_ALL, "") instead of resetlocale() on Windows when tearing down the locale test (#308).
- Fix for Sphinx warnings (#309).
- Addition of .cache, .idea, .pyd, .pdb to .gitignore (#310).

1.5.11 (2015-08-23)

- Remove packaging module requirement added in 1.5.10 (#305). Distutils can't parse versions using 'rc', but if we stick to 'a' and 'b' we will be fine.

1.5.10 (2015-08-22)

- Monkey patch affinity module by absolute reference (#299).
- Raise TopologicalError in relate() instead of crashing (#294, #295, #303).

1.5.9 (2015-05-27)

- Fix for 64 bit speedups compatibility (#274).

1.5.8 (2015-04-29)

- Setup file encoding bug fix (#254).
- Support for pyinstaller (#261).
- Major prepared geometry operation fix for Windows (#268, #269).
- Major fix for OS X binary wheel (#262).

1.5.7 (2015-03-16)

- Test and fix buggy error and notice handlers (#249).

1.5.6 (2015-02-02)

- Fix setup regression (#232, #234).
- SVG representation improvements (#233, #237).

1.5.5 (2015-01-20)

- MANIFEST changes to restore _geox.pxi (#231).

1.5.4 (2015-01-19)

- Fixed OS X binary wheel library load path (#224).

1.5.3 (2015-01-12)

- Fixed ownership and potential memory leak in polygonize (#223).
- Wider release of binary wheels for OS X.

1.5.2 (2015-01-04)

- Fail installation if GEOS dependency is not met, preventing update breakage (#218, #219).

1.5.1 (2014-12-04)

- Restore geometry hashing (#209).

1.5.0 (2014-12-02)

- Affine transformation speedups (#197).
- New == rich comparison (#195).
- Geometry collection constructor (#200).
- ops.snap() backed by GEOSSnap (#201).
- Clearer exceptions in cases of topological invalidity (#203).

1.4.4 (2014-11-02)

- Proper conversion of numpy float32 vals to coords (#186).

1.4.3 (2014-10-01)

- Fix for endianness bug in WKB writer (#174).

1.4.2 (2014-09-29)

- Fix bungled 1.4.1 release (#176).

1.4.1 (2014-09-23)

- Return of support for GEOS 3.2 (#176, #178).

1.4.0 (2014-09-08)

- SVG representations for IPython's inline image protocol.
- Efficient and fast vectorized contains().
- Change mitre_limit default to 5.0; raise ValueError with 0.0 (#139).
- Allow mix of tuples and Points in sped-up LineString ctor (#152).
- New STRtree class (#73).
- Add ops.nearest_points() (#147).
- Faster creation of geometric objects from others (cloning) (#165).
- Removal of tests from package.

1.3.3 (2014-07-23)

- Allow single-part geometries as argument to ops.cacaded_union() (#135).
- Support affine transformations of LinearRings (#112).

1.3.2 (2014-05-13)

- Let LineString() take a sequence of Points (#130).

1.3.1 (2014-04-22)

- More reliable proxy cleanup on exit (#106).
- More robust DLL loading on all platforms (#114).

1.3.0 (2013-12-31)

- Include support for Python 3.2 and 3.3 (#56), minimum version is now 2.6.
- Switch to GEOS WKT/WKB Reader/Writer API, with defaults changed to enable 3D output dimensions, and to 'trim' WKT output for GEOS >=3.3.0.
- Use GEOS version instead of GEOS C API version to determine library capabilities (#65).

1.2.19 (2013-12-30)

- Add buffering style options (#55).

1.2.18 (2013-07-23)

- Add shapely.ops.transform.
- Permit empty sequences in collection constructors (#49, #50).
- Individual polygons in MultiPolygon.__geo_interface__ are changed to tuples to match Polygon.__geo_interface__ (#51).
- Add shapely.ops.polygonize_full (#57).

1.2.17 (2013-01-27)

- Avoid circular import between wkt/wkb and geometry.base by moving calls to GEOS serializers to the latter module.
- Set _ndim when unpickling (issue #6).
- Don't install DLLs to Python's DLL directory (#37).
- Add affinity module of affine transformation (#31).
- Fix NameError that blocked installation with PyPy (#40, #41).

1.2.16 (2012-09-18)

- Add ops.unary_union function.
- Alias ops.cascaded_union to ops.unary_union when GEOS CAPI >= (1,7,0).
- Add geos_version_string attribute to shapely.geos.
- Ensure parent is set when child geometry is accessed.
- Generate _speedups.c using Cython when building from repo when missing, stale, or the build target is "sdist".
- The is_simple predicate of invalid, self-intersecting linear rings now returns `False`.
- Remove VERSION.txt from repo, it's now written by the distutils setup script with value of shapely.__version__.

1.2.15 (2012-06-27)

- Eliminate numerical sensitivity in a method chaining test (Debian bug #663210).
- Account for cascaded union of random buffered test points being a polygon or multipolygon (Debian bug #666655).
- Use Cython to build speedups if it is installed.
- Avoid stumbling over SVN revision numbers in GEOS C API version strings.

1.2.14 (2012-01-23)

- A geometry's coords property is now sliceable, yielding a list of coordinate values.
- Homogeneous collections are now sliceable, yielding a new collection of the same type.

1.2.13 (2011-09-16)

- Fixed errors in speedups on 32bit systems when GEOS references memory above 2GB.
- Add shapely.__version__ attribute.
- Update the manual.

1.2.12 (2011-08-15)

- Build Windows distributions with VC7 or VC9 as appropriate.
- More verbose report on failure to speed up.
- Fix for prepared geometries broken in 1.2.11.
- DO NOT INSTALL 1.2.11

1.2.11 (2011-08-04)

- Ignore AttributeError during exit.
- PyPy 1.5 support.
- Prevent operation on prepared geometry crasher (#12).
- Optional Cython speedups for Windows.
- Linux 3 platform support.

1.2.10 (2011-05-09)

- Add optional Cython speedups.
- Add is_cww predicate to LinearRing.
- Add function that forces orientation of Polygons.
- Disable build of speedups on Windows pending packaging work.

1.2.9 (2011-03-31)

- Remove extra glob import.
- Move examples to shapely.examples.
- Add box() constructor for rectangular polygons.
- Fix extraneous imports.

1.2.8 (2011-12-03)

- New parallel_offset method (#6).
- Support for Python 2.4.

1.2.7 (2010-11-05)

- Support for Windows eggs.

1.2.6 (2010-10-21)

- The geoms property of an empty collection yields [] instead of a ValueError (#3).
- The coords and geometry type sproperties have the same behavior as above.
- Ensure that z values carry through into products of operations (#4).

1.2.5 (2010-09-19)

- Stop distributing docs/_build.
- Include library fallbacks in test_dlls.py for linux platform.

1.2.4 (2010-09-09)

- Raise AttributeError when there's no backend support for a method.
- Raise OSError if libgeos_c.so (or variants) can't be found and loaded.
- Add geos_c DLL loading support for linux platforms where find_library doesn't work.

1.2.3 (2010-08-17)

- Add mapping function.
- Fix problem with GEOSisValidReason symbol for GEOS < 3.1.

1.2.2 (2010-07-23)

- Add representative_point method.

1.2.1 (2010-06-23)

- Fixed bounds of singular polygons.
- Added shapely.validation.explain_validity function (#226).

1.2 (2010-05-27)

- Final release.

1.2rc2 (2010-05-26)

- Add examples and tests to MANIFEST.in.
- Release candidate 2.

1.2rc1 (2010-05-25)

- Release candidate.

1.2b7 (2010-04-22)

- Memory leak associated with new empty geometry state fixed.

1.2b6 (2010-04-13)

- Broken GeometryCollection fixed.

1.2b5 (2010-04-09)

- Objects can be constructed from others of the same type, thereby making copies. Collections can be constructed from sequences of objects, also making copies.
- Collections are now iterators over their component objects.
- New code for manual figures, using the descartes package.

1.2b4 (2010-03-19)

- Adds support for the "sunos5" platform.

1.2b3 (2010-02-28)

- Only provide simplification implementations for GEOS C API >= 1.5.

1.2b2 (2010-02-19)

- Fix cascaded_union bug introduced in 1.2b1 (#212).

1.2b1 (2010-02-18)

- Update the README. Remove cruft from setup.py. Add some version 1.2 metadata regarding required Python version (>=2.5,<3) and external dependency (libgeos_c >= 3.1).

1.2a6 (2010-02-09)

- Add accessor for separate arrays of X and Y values (#210).

TODO: fill gap here

1.2a1 (2010-01-20)

- Proper prototyping of WKB writer, and avoidance of errors on 64-bit systems (#191).
- Prototype libgeos_c functions in a way that lets py2exe apps import shapely (#189).

1.2 Branched (2009-09-19)

1.0.12 (2009-04-09)

- Fix for references held by topology and predicate descriptors.

1.0.11 (2008-11-20)

- Work around bug in GEOS 2.2.3, GEOSCoordSeq_getOrdinate not exported properly (#178).

1.0.10 (2008-11-17)

- Fixed compatibility with GEOS 2.2.3 that was broken in 1.0.8 release (#176).

1.0.9 (2008-11-16)

- Find and load MacPorts libgeos.

1.0.8 (2008-11-01)

- Fill out GEOS function result and argument types to prevent faults on a 64-bit arch.

1.0.7 (2008-08-22)

- Polygon rings now have the same dimensions as parent (#168).
- Eliminated reference cycles in polygons (#169).

1.0.6 (2008-07-10)

- Fixed adaptation of multi polygon data.
- Raise exceptions earlier from binary predicates.
- Beginning distributing new windows DLLs (#166).

1.0.5 (2008-05-20)

- Added access to GEOS polygonizer function.
- Raise exception when insufficient coordinate tuples are passed to LinearRing constructor (#164).

1.0.4 (2008-05-01)

- Disentangle Python and topological equality (#163).
- Add shape(), a factory that copies coordinates from a geo interface provider. To be used instead of asShape() unless you really need to store coordinates outside shapely for efficient use in other code.
- Cache GEOS geometries in adapters (#163).

1.0.3 (2008-04-09)

- Do not release GIL when calling GEOS functions (#158).
- Prevent faults when chaining multiple GEOS operators (#159).

1.0.2 (2008-02-26)

- Fix loss of dimensionality in polygon rings (#155).

1.0.1 (2008-02-08)

- Allow chaining expressions involving coordinate sequences and geometry parts (#151).
- Protect against abnormal use of coordinate accessors (#152).
- Coordinate sequences now implement the numpy array protocol (#153).

1.0 (2008-01-18)

- Final release.

1.0 RC2 (2008-01-16)

- Added temporary solution for #149.

1.0 RC1 (2008-01-14)

- First release candidate

5.6 Geometry

Shapely geometry classes, such as `shapely.Point`, are the central data types in Shapely. Each geometry class extends the `shapely.Geometry` base class, which is a container of the underlying GEOS geometry object, to provide geometry type-specific attributes and behavior. The `Geometry` object keeps track of the underlying GEOS geometry 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 Shapely operation will result in a new object being returned.

5.6.1 Geometry types

Point(*args)	A geometry type that represents a single coordinate with x,y and possibly z values.
LineString([coordinates])	A geometry type composed of one or more line segments.
LinearRing([coordinates])	A geometry type composed of one or more line segments that forms a closed loop.
Polygon([shell, holes])	A geometry type representing an area that is enclosed by a linear ring.
MultiPoint([points])	A collection of one or more Points.
MultiLineString([lines])	A collection of one or more LineStrings.
MultiPolygon([polygons])	A collection of one or more Polygons.
GeometryCollection([geoms])	A collection of one or more geometries that may contain more than one type of geometry.

shapely.Point

class Point(**args*)

A geometry type that represents a single coordinate with x,y and possibly z values.

A point is a zero-dimensional feature and has zero length and zero area.

Parameters

args
[float, or sequence of floats] The coordinates can either be passed as a single parameter, or as individual float values using multiple parameters:

1) 1 parameter: a sequence or array-like of with 2 or 3 values.

2) 2 or 3 parameters (float): x, y, and possibly z.

Examples

Constructing the Point using separate parameters for x and y:

```
>>> p = Point(1.0, -1.0)
```

Constructing the Point using a list of x, y coordinates:

```
>>> p = Point([1.0, -1.0])
>>> print(p)
POINT (1 -1)
>>> p.y
-1.0
>>> p.x
1.0
```

Attributes

x, y, z
[float] Coordinate values

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat')

result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *fill_color=None*, *opacity=None*)

Returns SVG circle element for the Point geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG circle diameter. Default is 1.

fill_color
[str, optional] Hex string for fill color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.6

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property x

Return x coordinate.

property xy

Separate arrays of X and Y coordinate values

Example:

```
>>> x, y = Point(0, 0).xy
>>> list(x)
[0.0]
>>> list(y)
[0.0]
```

property y

Return y coordinate.

property z

Return z coordinate.

shapely.LineString

class LineString(*coordinates=None*)

A geometry type composed of one or more line segments.

A LineString is a one-dimensional feature and has a non-zero length but zero area. It may approximate a curve and need not be straight. Unlike a LinearRing, a LineString is not closed.

Parameters

coordinates
[sequence] A sequence of (x, y, [,z]) numeric coordinate pairs or triples, or an array-like with shape (N, 2) or (N, 3). Also can be a sequence of Point objects.

Examples

Create a LineString with two segments

```
>>> a = LineString([[0, 0], [1, 0], [1, 1]])
>>> a.length
2.0
```

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

offset_curve(*distance*, *quad_segs=16*, *join_style=BufferJoinStyle.round*, *mitre_limit=5.0*)

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

The side is determined by the sign of the *distance* parameter (negative for right side offset, positive for left side offset). The resolution of the buffer around each vertex of the object increases by increasing the *quad_segs* keyword parameter.

The join style is for outside corners between line segments. Accepted values are JOIN_STYLE.round (1), JOIN_STYLE.mitre (2), and JOIN_STYLE.bevel (3).

The mitre ratio limit is used for very sharp corners. It is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend far beyond the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

Note: the behaviour regarding orientation of the resulting line depends on the GEOS version. With GEOS < 3.11, the line retains the same direction for a left offset (positive distance) or has reverse direction for a right offset (negative distance), and this behaviour was documented as such in previous Shapely versions. Starting with GEOS 3.11, the function tries to preserve the orientation of the original line.

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

parallel_offset(*distance*, *side='right'*, *resolution=16*, *join_style=BufferJoinStyle.round*, *mitre_limit=5.0*)

Alternative method to *offset_curve()* method.

Older alternative method to the *offset_curve()* method, but uses `resolution` instead of `quad_segs` and a `side` keyword ('left' or 'right') instead of sign of the distance. This method is kept for backwards compatibility for now, but is is recommended to use *offset_curve()* instead.

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *stroke_color=None*, *opacity=None*)

Returns SVG polyline element for the LineString geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG stroke-width. Default is 1.

stroke_color
[str, optional] Hex string for stroke color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.8

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

Example:

```
>>> x, y = LineString([(0, 0), (1, 1)]).xy
>>> list(x)
[0.0, 1.0]
>>> list(y)
[0.0, 1.0]
```

shapely.LinearRing

class LinearRing(*coordinates=None*)

A geometry type composed of one or more line segments that forms a closed loop.

A LinearRing is a closed, one-dimensional feature. A LinearRing that crosses itself or touches itself at a single point is invalid and operations on it may fail.

Parameters

coordinates
[sequence] A sequence of (x, y [,z]) numeric coordinate pairs or triples, or an array-like with shape (N, 2) or (N, 3). Also can be a sequence of Point objects.

Notes

Rings are automatically closed. There is no need to specify a final coordinate pair identical to the first.

Examples

Construct a square ring.

```
>>> ring = LinearRing( ((0, 0), (0, 1), (1 ,1 ), (1 , 0)) )
>>> ring.is_closed
True
>>> list(ring.coords)
[(0.0, 0.0), (0.0, 1.0), (1.0, 1.0), (1.0, 0.0), (0.0, 0.0)]
>>> ring.length
4.0
```

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent

unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_ccw

True is the ring is oriented counter clock-wise

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

offset_curve(*distance*, *quad_segs=16*, *join_style=BufferJoinStyle.round*, *mitre_limit=5.0*)

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

The side is determined by the sign of the *distance* parameter (negative for right side offset, positive for left side offset). The resolution of the buffer around each vertex of the object increases by increasing the *quad_segs* keyword parameter.

The join style is for outside corners between line segments. Accepted values are JOIN_STYLE.round (1), JOIN_STYLE.mitre (2), and JOIN_STYLE.bevel (3).

The mitre ratio limit is used for very sharp corners. It is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend far beyond the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

Note: the behaviour regarding orientation of the resulting line depends on the GEOS version. With GEOS < 3.11, the line retains the same direction for a left offset (positive distance) or has reverse direction for a right offset (negative distance), and this behaviour was documented as such in previous Shapely versions. Starting with GEOS 3.11, the function tries to preserve the orientation of the original line.

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

parallel_offset(*distance*, *side='right'*, *resolution=16*, *join_style=BufferJoinStyle.round*, *mitre_limit=5.0*)

Alternative method to *offset_curve()* method.

Older alternative method to the *offset_curve()* method, but uses `resolution` instead of `quad_segs` and a `side` keyword ('left' or 'right') instead of sign of the distance. This method is kept for backwards compatibility for now, but is is recommended to use *offset_curve()* instead.

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *stroke_color=None*, *opacity=None*)

Returns SVG polyline element for the LineString geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG stroke-width. Default is 1.

stroke_color
[str, optional] Hex string for stroke color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.8

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

Example:

```
>>> x, y = LineString([(0, 0), (1, 1)]).xy
>>> list(x)
[0.0, 1.0]
>>> list(y)
[0.0, 1.0]
```

shapely.Polygon

class Polygon(*shell=None*, *holes=None*)

A geometry type representing an area that is enclosed by a linear ring.

A polygon is a two-dimensional feature and has a non-zero area. It may have one or more negative-space "holes" which are also bounded by linear rings. If any rings cross each other, the feature is invalid and operations on it may fail.

Parameters

shell
[sequence] A sequence of (x, y [,z]) numeric coordinate pairs or triples, or an array-like with shape (N, 2) or (N, 3). Also can be a sequence of Point objects.

holes
[sequence] A sequence of objects which satisfy the same requirements as the shell parameters above

Examples

Create a square polygon with no holes

```
>>> coords = ((0., 0.), (0., 1.), (1., 1.), (1., 0.), (0., 0.))
>>> polygon = Polygon(coords)
>>> polygon.area
1.0
```

Attributes

exterior
[LinearRing] The ring which bounds the positive space of the polygon.

interiors
[sequence] A sequence of rings which bound all existing holes.

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent

unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

`contains_properly`(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

`property convex_hull`

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

`property coords`

Access to geometry's coordinates (CoordinateSequence)

`covered_by`(*other*)

Returns True if the geometry is covered by the other, else False

`covers`(*other*)

Returns True if the geometry covers the other, else False

`crosses`(*other*)

Returns True if the geometries cross, else False

`difference`(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

`disjoint`(*other*)

Returns True if geometries are disjoint, else False

`distance`(*other*)

Unitless distance to other geometry (float)

`dwithin`(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

`property envelope`

A figure that envelopes the geometry

`equals`(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

classmethod from_bounds(*xmin*, *ymin*, *xmax*, *ymax*)

Construct a *Polygon()* from spatial bounds.

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *fill_color=None*, *opacity=None*)

Returns SVG path element for the Polygon geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG stroke-width. Default is 1.

fill_color
[str, optional] Hex string for fill color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.6

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

shapely.MultiPoint

class MultiPoint(*points=None*)

A collection of one or more Points.

A MultiPoint has zero area and zero length.

Parameters

points
[sequence] A sequence of Points, or a sequence of (x, y [,z]) numeric coordinate pairs or triples, or an array-like of shape (N, 2) or (N, 3).

Examples

Construct a MultiPoint containing two Points

```
>>> from shapely import Point
>>> ob = MultiPoint([[0.0, 0.0], [1.0, 2.0]])
>>> len(ob.geoms)
2
>>> type(ob.geoms[0]) == Point
True
```

Attributes

geoms
[sequence] A sequence of Points

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat')

result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *fill_color=None*, *opacity=None*)

Returns a group of SVG circle elements for the MultiPoint geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG circle diameters. Default is 1.

fill_color
[str, optional] Hex string for fill color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.6

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

shapely.MultiLineString

class MultiLineString(*lines=None*)

A collection of one or more LineStrings.

A MultiLineString has non-zero length and zero area.

Parameters

lines
[sequence] A sequence LineStrings, or a sequence of line-like coordinate sequences or array-likes (see accepted input for LineString).

Examples

Construct a MultiLineString containing two LineStrings.

```
>>> lines = MultiLineString([[[0, 0], [1, 2]], [[4, 4], [5, 6]]])
```

Attributes

geoms
[sequence] A sequence of LineStrings

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *stroke_color=None*, *opacity=None*)

Returns a group of SVG polyline elements for the LineString geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG stroke-width. Default is 1.

stroke_color
[str, optional] Hex string for stroke color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.8

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

shapely.MultiPolygon

class MultiPolygon(*polygons=None*)

A collection of one or more Polygons.

If component polygons overlap the collection is invalid and some operations on it may fail.

Parameters

polygons
[sequence] A sequence of Polygons, or a sequence of (shell, holes) tuples where shell is the sequence representation of a linear ring (see LinearRing) and holes is a sequence of such linear rings.

Examples

Construct a MultiPolygon from a sequence of coordinate tuples

```
>>> from shapely import Polygon
>>> ob = MultiPolygon([
...     (
...     ((0.0, 0.0), (0.0, 1.0), (1.0, 1.0), (1.0, 0.0)),
...     [((0.1,0.1), (0.1,0.2), (0.2,0.2), (0.2,0.1))]
...     )
... ])
>>> len(ob.geoms)
1
>>> type(ob.geoms[0]) == Polygon
True
```

Attributes

geoms
[sequence] A sequence of *Polygon* instances

almost_equals(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property area

Unitless area of the geometry (float)

property boundary

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

property bounds

Returns minimum bounding region (minx, miny, maxx, maxy)

buffer(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *fill_color=None*, *opacity=None*)

Returns group of SVG path elements for the MultiPolygon geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG stroke-width. Default is 1.

fill_color
[str, optional] Hex string for fill color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

opacity
[float] Float number between 0 and 1 for color opacity. Default value is 0.6

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

shapely.GeometryCollection

class GeometryCollection(*geoms=None*)

A collection of one or more geometries that may contain more than one type of geometry.

Parameters

geoms
[list] A list of shapely geometry instances, which may be of varying geometry types.

Examples

Create a GeometryCollection with a Point and a LineString

```
>>> from shapely import LineString, Point
>>> p = Point(51, -1)
>>> l = LineString([(52, -1), (49, 2)])
>>> gc = GeometryCollection([p, l])
```

Attributes

geoms
[sequence] A sequence of Shapely geometry instances

`almost_equals`(*other*, *decimal=6*)

True if geometries are equal at all coordinates to a specified decimal place.

Deprecated since version 1.8.0: The 'almost_equals()' method is deprecated and will be removed in Shapely 2.1 because the name is confusing. The 'equals_exact()' method should be used instead.

Refers to approximate coordinate equality, which requires coordinates to be approximately equal and in the same order for all components of a geometry.

Because of this it is possible for "equals()" to be True for two geometries and "almost_equals()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

`property area`

Unitless area of the geometry (float)

`property boundary`

Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

`property bounds`

Returns minimum bounding region (minx, miny, maxx, maxy)

`buffer`(*distance*, *quad_segs=16*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Get a geometry that represents all points within a distance of this geometry.

A positive distance produces a dilation, a negative distance an erosion. A very small or zero distance may sometimes be used to "tidy" a polygon.

Parameters

distance
[float] The distance to buffer around the object.

resolution
[int, optional] The resolution of the buffer around each vertex of the object.

quad_segs
[int, optional] Sets the number of line segments used to approximate an angle fillet.

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.ROUND ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, optional] The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

single_side
[bool, optional] The side used is determined by the sign of the buffer distance:

a positive distance indicates the left-hand side a negative distance indicates the right-hand side

The single-sided buffer of point geometries is the same as the regular buffer. The End Cap Style for single-sided buffers is always ignored, and forced to the equivalent of CAP_FLAT.

quadsegs
[int, optional] Deprecated alias for *quad_segs*.

Returns

Geometry

Notes

The return value is a strictly two-dimensional geometry. All Z coordinates of the original geometry will be ignored.

Examples

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
```

16-gon approx of a unit radius circle:

```
>>> g.buffer(1.0).area
3.1365484905459...
```

128-gon approximation:

```
>>> g.buffer(1.0, 128).area
3.141513801144...
```

triangle approximation:

```
>>> g.buffer(1.0, 3).area
3.0
>>> list(g.buffer(1.0, cap_style=BufferCapStyle.square).exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style=BufferCapStyle.square).area
4.0
```

property centroid

Returns the geometric center of the object

contains(*other*)

Returns True if the geometry contains the other, else False

contains_properly(*other*)

Returns True if the geometry completely contains the other, with no common boundary points, else False

Refer to *shapely.contains_properly* for full documentation.

property convex_hull

Imagine an elastic band stretched around the geometry: that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

property coords

Access to geometry's coordinates (CoordinateSequence)

covered_by(*other*)

Returns True if the geometry is covered by the other, else False

covers(*other*)

Returns True if the geometry covers the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*, *grid_size=None*)

Returns the difference of the geometries.

Refer to *shapely.difference* for full documentation.

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dwithin(*other*, *distance*)

Returns True if geometry is within a given distance from the other, else False.

Refer to *shapely.dwithin* for full documentation.

property envelope

A figure that envelopes the geometry

equals(*other*)

Returns True if geometries are equal, else False.

This method considers point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other)).

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals(
...     LineString([(0, 0), (1, 1), (2, 2)])
... )
True
```

equals_exact(*other*, *tolerance*)

True if geometries are equal to within a specified tolerance.

Parameters

other
[BaseGeometry] The other geometry object in this comparison.

tolerance
[float] Absolute tolerance in the same units as coordinates.

This method considers coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry.
Because of this it is possible for "equals()" to be True for two geometries and "equals_exact()" to be False.

Returns

bool

Examples

```
>>> LineString(
...     [(0, 0), (2, 2)]
... ).equals_exact(
...     LineString([(0, 0), (1, 1), (2, 2)]),
...     1e-6
... )
False
```

property geom_type

Name of the geometry's type, such as 'Point'

property has_z

True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

hausdorff_distance(*other*)

Unitless hausdorff distance to other geometry (float)

interpolate(*distance*, *normalized=False*)

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *line_interpolate_point*.

intersection(*other*, *grid_size=None*)

Returns the intersection of the geometries.

Refer to *shapely.intersection* for full documentation.

intersects(*other*)

Returns True if geometries intersect, else False

property is_closed

True if the geometry is closed, else False

Applicable only to 1-D geometries.

property is_empty

True if the set of points in this geometry is empty, else False

property is_ring

True if the geometry is a closed ring, else False

property is_simple

True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

property is_valid

True if the geometry is valid (definition depends on sub-class), else False

property length

Unitless length of the geometry (float)

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

Return a point at the specified distance along a linear geometry

Negative length values are taken as measured in the reverse direction from the end of the geometry. Out-of-range index values are handled by clamping them to the valid range of values. If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

Alias of *interpolate*.

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

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *project*.

property minimum_clearance

Unitless distance by which a node could be moved to produce an invalid geometry (float)

property minimum_rotated_rectangle

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike *envelope* this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *oriented_envelope*.

normalize()

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*).

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(3, 3), (2, 2)]])
>>> line.normalize()
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

property oriented_envelope

Returns the oriented envelope (minimum rotated rectangle) that encloses the geometry.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Alias of *minimum_rotated_rectangle*.

overlaps(*other*)

Returns True if geometries overlap, else False

point_on_surface()

Returns a point guaranteed to be within the object, cheaply.

Alias of *representative_point*.

project(*other*, *normalized=False*)

Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

Alias of *line_locate_point*.

relate(*other*)

Returns the DE-9IM intersection matrix for the two geometries (string)

relate_pattern(*other*, *pattern*)

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

representative_point()

Returns a point guaranteed to be within the object, cheaply.

Alias of *point_on_surface*.

reverse()

Returns a copy of this geometry with the order of coordinates reversed.

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

Points are unchanged.

See also:

is_ccw
Checks if a geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (1, 2)]).reverse()
<LINESTRING (1 2, 0 0)>
>>> Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]).reverse()
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

segmentize(*max_segment_length*)

Adds vertices to line segments based on maximum segment length.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

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

Examples

```
>>> from shapely import LineString, Polygon
>>> LineString([(0, 0), (0, 10)]).segmentize(max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]).segmentize(max_
↪segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
```

simplify(*tolerance*, *preserve_topology=True*)

Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

svg(*scale_factor=1.0*, *color=None*)

Returns a group of SVG elements for the multipart geometry.

Parameters

scale_factor
[float] Multiplication factor for the SVG stroke-width. Default is 1.

color
[str, optional] Hex string for stroke or fill color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

symmetric_difference(*other*, *grid_size=None*)

Returns the symmetric difference of the geometries.

Refer to *shapely.symmetric_difference* for full documentation.

touches(*other*)

Returns True if geometries touch, else False

union(*other*, *grid_size=None*)

Returns the union of the geometries.

Refer to *shapely.union* for full documentation.

within(*other*)

Returns True if geometry is within the other, else False

property wkb

WKB representation of the geometry

property wkb_hex

WKB hex representation of the geometry

property wkt

WKT representation of the geometry

property xy

Separate arrays of X and Y coordinate values

5.6.2 Construction

Geometries can be constructed directly using Shapely geometry classes:

```
>>> from shapely import Point, LineString
>>> Point(5.2, 52.1)
<POINT (5.2 52.1)>
>>> LineString([(0, 0), (1, 2)])
<LINESTRING (0 0, 1 2)>
```

Geometries can also be constructed from a WKT (Well-Known Text) or WKB (Well-Known Binary) representation:

```
>>> from shapely import from_wkb, from_wkt
>>> from_wkt("POINT (5.2 52.1)")
<POINT (5.2 52.1)>
>>> from_wkb(b"\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\
↪xf0?")
<POINT (1 1)>
```

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

5.6.3 Pickling

Geometries can be serialized using pickle:

```
>>> import pickle
>>> from shapely import Point
>>> pickled = pickle.dumps(Point(1, 1))
>>> pickle.loads(pickled)
<POINT (1 1)>
```

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

5.6.4 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. Shapely generates this hash from the WKB representation. Therefore, geometries are equal if and only if their WKB representations are equal.

```
>>> from shapely import Point
>>> point_1 = Point(5.2, 52.1)
>>> point_2 = Point(1, 1)
>>> point_3 = Point(5.2, 52.1)
>>> {point_1, point_2, point_3}
{<POINT (1 1)>, <POINT (5.2 52.1)>}
```

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

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

```
>>> point_1 == point_2
False
>>> point_1 == point_3
True
>>> point_1 != point_2
True
```

5.6.5 Formatting

Geometries can be formatted to strings using properties, functions, or a Python format specification.

The most convenient is to use `.wkb_hex` and `.wkt` properties.

```
>>> from shapely import Point, to_wkb, to_wkt, to_geojson
>>> pt = Point(-169.910918, -18.997564)
>>> pt.wkb_hex
0101000000CF6A813D263D65C0BDAAB35A60FF32C0
```

(continues on next page)

(continued from previous page)

```
>>> pt.wkt
POINT (-169.910918 -18.997564)
```

More output options can be found using using *to_wkb()*, *to_wkt()*, and *to_geojson()* functions.

```
>>> to_wkb(pt, hex=True, byte_order=0)
000000000101C0653D263D816ACFC032FF605AB3AABD
>>> to_wkt(pt, rounding_precision=3)
POINT (-169.911 -18.998)
>>> print(to_geojson(pt, indent=2))
{
  "type": "Point",
  "coordinates": [
    -169.910918,
    -18.997564
  ]
}
```

A format specification may also be used to control the format and precision.

```
>>> print(f"Cave near {pt:.3f}")
Cave near POINT (-169.911 -18.998)
>>> print(f"or hex-encoded as {pt:x}")
or hex-encoded as 0101000000cf6a813d263d65c0bdaab35a60ff32c0
```

Shapely has a format specification inspired from Python's Format Specification Mini-Language, described next.

Semantic for format specification

```
format_spec   ::=   [0][.precision][type]
precision     ::=   digit+
digit         ::=   "0"..."9"
type          ::=   "f" | "F" | "g" | "G" | "x" | "X"
```

Format types `'f'` and `'F'` are to use a fixed-point notation, which is activated by setting GEOS' trim option off. The upper case variant converts `nan` to `NAN` and `inf` to `INF`.

Format types `'g'` and `'G'` are to use a "general format", where unnecessary digits are trimmed. This notation is activated by setting GEOS' trim option on. The upper case variant is similar to `'F'`, and may also display an upper-case `"E"` if scientific notation is required. Note that this representation may be different for GEOS 3.10.0 and later, which does not use scientific notation.

For numeric outputs `'f'` and `'g'`, the precision is optional, and if not specified, rounding precision will be disabled showing full precision.

Format types `'x'` and `'X'` show a hex-encoded string representation of WKB or Well-Known Binary, with the case of the output matched the case of the format type character.

5.7 Geometry properties

GeometryType(value)	The enumeration of GEOS geometry types
get_type_id(geometry, **kwargs)	Returns the type ID of a geometry.
get_dimensions(geometry, **kwargs)	Returns the inherent dimensionality of a geometry.
get_coordinate_dimension(geometry, **kwargs)	Returns the dimensionality of the coordinates in a geometry (2 or 3).
get_num_coordinates(geometry, **kwargs)	Returns the total number of coordinates in a geometry.
get_srid(geometry, **kwargs)	Returns the SRID of a geometry.
set_srid(geometry, srid, **kwargs)	Returns a geometry with its SRID set.
get_x(point, **kwargs)	Returns the x-coordinate of a point
get_y(point, **kwargs)	Returns the y-coordinate of a point
get_z(point, **kwargs)	Returns the z-coordinate of a point.
get_exterior_ring(geometry, **kwargs)	Returns the exterior ring of a polygon.
get_num_points(geometry, **kwargs)	Returns number of points in a linestring or linearring.
get_num_interior_rings(geometry, **kwargs)	Returns number of internal rings in a polygon
get_num_geometries(geometry, **kwargs)	Returns number of geometries in a collection.
get_point(geometry, index, **kwargs)	Returns the nth point of a linestring or linearring.
get_interior_ring(geometry, index, **kwargs)	Returns the nth interior ring of a polygon.
get_geometry(geometry, index, **kwargs)	Returns the nth geometry from a collection of geometries.
get_parts(geometry[, return_index])	Gets parts of each GeometryCollection or Multi* geometry object; returns a copy of each geometry in the GeometryCollection or Multi* geometry object.
get_rings(geometry[, return_index])	Gets rings of Polygon geometry object.
get_precision(geometry, **kwargs)	Get the precision of a geometry.
set_precision(geometry, grid_size[, mode])	Returns geometry with the precision set to a precision grid size.
force_2d(geometry, **kwargs)	Forces the dimensionality of a geometry to 2D.
force_3d(geometry[, z])	Forces the dimensionality of a geometry to 3D.

5.7.1 shapely.GeometryType

class GeometryType(*value*)

The enumeration of GEOS geometry types

5.7.2 shapely.get_type_id

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

Returns the type ID of a geometry.

- None (missing) is -1
- POINT is 0
- LINESTRING is 1
- LINEARRING is 2
- POLYGON is 3
- MULTIPOINT is 4

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

GeometryType

Examples

```
>>> from shapely import LineString, Point
>>> get_type_id(LineString([(0, 0), (1, 1), (2, 2), (3, 3)]))
1
>>> get_type_id([Point(1, 2), Point(2, 3)]).tolist()
[0, 0]
```

5.7.3 shapely.get_dimensions

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

Returns the inherent dimensionality of a geometry.

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, Point, Polygon
>>> point = Point(0, 0)
>>> get_dimensions(point)
0
>>> polygon = Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)])
>>> get_dimensions(polygon)
2
>>> get_dimensions(GeometryCollection([point, polygon]))
2
```

(continues on next page)

(continued from previous page)

```
>>> get_dimensions(GeometryCollection([]))
-1
>>> get_dimensions(None)
-1
```

5.7.4 shapely.get_coordinate_dimension

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

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

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import Point
>>> get_coordinate_dimension(Point(0, 0))
2
>>> get_coordinate_dimension(Point(0, 0, 1))
3
>>> get_coordinate_dimension(None)
-1
>>> get_coordinate_dimension(Point(0, 0, float("nan")))
2
```

5.7.5 shapely.get_num_coordinates

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

Returns the total number of coordinates in a geometry.

Returns 0 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, LineString, Point
>>> point = Point(0, 0)
>>> get_num_coordinates(point)
1
>>> get_num_coordinates(Point(0, 0, 0))
1
>>> line = LineString([(0, 0), (1, 1)])
>>> get_num_coordinates(line)
2
>>> get_num_coordinates(GeometryCollection([point, line]))
3
>>> get_num_coordinates(None)
0
```

5.7.6 shapely.get_srid

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

Returns the SRID of a geometry.

Returns -1 for not-a-geometry values.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

set_srid

Examples

```
>>> from shapely import Point
>>> point = Point(0, 0)
>>> get_srid(point)
0
>>> with_srid = set_srid(point, 4326)
>>> get_srid(with_srid)
4326
```

5.7.7 shapely.set_srid

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

Returns a geometry with its SRID set.

Parameters

geometry
[Geometry or array_like]

srid
[int]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_srid

Examples

```
>>> from shapely import Point
>>> point = Point(0, 0)
>>> get_srid(point)
0
>>> with_srid = set_srid(point, 4326)
>>> get_srid(with_srid)
4326
```

5.7.8 shapely.get_x

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

Returns the x-coordinate of a point

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_y, *get_z*

Examples

```
>>> from shapely import MultiPoint, Point
>>> get_x(Point(1, 2))
1.0
>>> get_x(MultiPoint([(1, 1), (1, 2)]))
nan
```

5.7.9 shapely.get_y

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

Returns the y-coordinate of a point

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_x, *get_z*

Examples

```
>>> from shapely import MultiPoint, Point
>>> get_y(Point(1, 2))
2.0
>>> get_y(MultiPoint([(1, 1), (1, 2)]))
nan
```

5.7.10 shapely.get_z

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

Returns the z-coordinate of a point.

Note: 'get_z' requires at least GEOS 3.7.0.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_x, *get_y*

Examples

```
>>> from shapely import MultiPoint, Point
>>> get_z(Point(1, 2, 3))
3.0
>>> get_z(Point(1, 2))
nan
>>> get_z(MultiPoint([(1, 1, 1), (2, 2, 2)]))
nan
```

5.7.11 shapely.get_exterior_ring

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

Returns the exterior ring of a polygon.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_interior_ring

Examples

```
>>> from shapely import Point, Polygon
>>> get_exterior_ring(Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)]))
<LINEARRING (0 0, 0 10, 10 10, 10 0, 0 0)>
>>> get_exterior_ring(Point(1, 1)) is None
True
```

5.7.12 shapely.get_num_points

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

Returns number of points in a linestring or linearring.

Returns 0 for not-a-geometry values.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_point
get_num_geometries

Examples

```
>>> from shapely import LineString, MultiPoint
>>> get_num_points(LineString([(0, 0), (1, 1), (2, 2), (3, 3)]))
4
>>> get_num_points(MultiPoint([(0, 0), (1, 1), (2, 2), (3, 3)]))
0
>>> get_num_points(None)
0
```

5.7.13 shapely.get_num_interior_rings

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

Returns number of internal rings in a polygon

Returns 0 for not-a-geometry values.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_exterior_ring
get_interior_ring

Examples

```
>>> from shapely import Point, Polygon
>>> polygon = Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)])
>>> get_num_interior_rings(polygon)
0
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> get_num_interior_rings(polygon_with_hole)
1
>>> get_num_interior_rings(Point(0, 0))
0
>>> get_num_interior_rings(None)
0
```

5.7.14 shapely.get_num_geometries

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

Returns number of geometries in a collection.

Returns 0 for not-a-geometry values.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_num_points
get_geometry

Examples

```
>>> from shapely import MultiPoint, Point
>>> get_num_geometries(MultiPoint([(0, 0), (1, 1), (2, 2), (3, 3)]))
4
>>> get_num_geometries(Point(1, 1))
1
>>> get_num_geometries(None)
0
```

5.7.15 shapely.get_point

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

Returns the nth point of a linestring or linearring.

Parameters

geometry
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_num_points

Examples

```
>>> from shapely import LinearRing, LineString, MultiPoint, Point
>>> line = LineString([(0, 0), (1, 1), (2, 2), (3, 3)])
>>> get_point(line, 1)
<POINT (1 1)>
>>> get_point(line, -2)
<POINT (2 2)>
>>> get_point(line, [0, 3]).tolist()
[<POINT (0 0)>, <POINT (3 3)>]
```

The functcion works the same for LinearRing input:

```
>>> get_point(LinearRing([(0, 0), (1, 1), (2, 2), (0, 0)]), 1)
<POINT (1 1)>
```

For non-linear geometries it returns None:

```
>>> get_point(MultiPoint([(0, 0), (1, 1), (2, 2), (3, 3)]), 1) is None
True
>>> get_point(Point(1, 1), 0) is None
True
```

5.7.16 shapely.get_interior_ring

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

Returns the nth interior ring of a polygon.

Parameters

geometry
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_exterior_ring
get_num_interior_rings

Examples

```
>>> from shapely import Point, Polygon
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> get_interior_ring(polygon_with_hole, 0)
<LINEARRING (2 2, 2 4, 4 4, 4 2, 2 2)>
```

(continues on next page)

(continued from previous page)

```
>>> get_interior_ring(polygon_with_hole, 1) is None
True
>>> polygon = Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)])
>>> get_interior_ring(polygon, 0) is None
True
>>> get_interior_ring(Point(0, 0), 0) is None
True
```

5.7.17 shapely.get_geometry

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

Returns the nth geometry from a collection of geometries.

Parameters

geometry
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_num_geometries, *get_parts*

Notes

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

Examples

```
>>> from shapely import Point, MultiPoint
>>> multipoint = MultiPoint([(0, 0), (1, 1), (2, 2), (3, 3)])
>>> get_geometry(multipoint, 1)
<POINT (1 1)>
>>> get_geometry(multipoint, -1)
<POINT (3 3)>
>>> get_geometry(multipoint, 5) is None
True
>>> get_geometry(Point(1, 1), 0)
<POINT (1 1)>
>>> get_geometry(Point(1, 1), 1) is None
True
```

5.7.18 shapely.get_parts

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

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

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

Parameters

geometry
[Geometry or array_like]

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

Returns

ndarray of parts or tuple of (parts, indexes)

See also:

get_geometry, *get_rings*

Examples

```
>>> from shapely import MultiPoint
>>> get_parts(MultiPoint([(0, 1), (2, 3)])).tolist()
[<POINT (0 1)>, <POINT (2 3)>]
>>> parts, index = get_parts([MultiPoint([(0, 1)]), MultiPoint([(4, 5), (6, 7)])],␣
↪return_index=True)
>>> parts.tolist()
[<POINT (0 1)>, <POINT (4 5)>, <POINT (6 7)>]
>>> index.tolist()
[0, 1, 1]
```

5.7.19 shapely.get_rings

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

Gets rings of Polygon geometry object.

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

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

Parameters

geometry
[Geometry or array_like]

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

Returns

ndarray of rings or tuple of (rings, indexes)

See also:

get_exterior_ring, *get_interior_ring*, *get_parts*

Examples

```
>>> from shapely import Polygon
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> get_rings(polygon_with_hole).tolist()
[<LINEARRING (0 0, 0 10, 10 10, 10 0, 0 0)>,
 <LINEARRING (2 2, 2 4, 4 4, 4 2, 2 2)>]
```

With `return_index=True`:

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

5.7.20 shapely.get_precision

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

Get the precision of a geometry.

Note: 'get_precision' requires at least GEOS 3.6.0.

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

Returns NaN for not-a-geometry values.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

set_precision

Examples

```
>>> from shapely import Point
>>> point = Point(1, 1)
>>> get_precision(point)
0.0
>>> geometry = set_precision(point, 1.0)
>>> get_precision(geometry)
1.0
>>> get_precision(None)
nan
```

5.7.21 shapely.set_precision

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

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

Note: 'set_precision' requires at least GEOS 3.6.0.

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

Coordinates will be rounded if a precision grid is less precise than the input geometry. Duplicated vertices will be dropped from lines and polygons for grid sizes greater than 0. Line and polygon geometries may collapse to empty geometries if all vertices are closer together than grid_size. Z values, if present, will not be modified.

Note: subsequent operations will always be performed in the precision of the geometry with higher precision (smaller "grid_size"). That same precision will be attached to the operation outputs.

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

Returns None if geometry is None.

Parameters

geometry
[Geometry or array_like]

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

mode
[{'valid_output', 'pointwise', 'keep_collapsed'}, default 'valid_output'] This parameter determines how to handle invalid output geometries. There are three modes:

1. *'valid_output'* (default): The output is always valid. Collapsed geometry elements (including both polygons and lines) are removed. Duplicate vertices are removed.

2. *'pointwise'*: Precision reduction is performed pointwise. Output geometry may be invalid due to collapse or self-intersection. Duplicate vertices are not removed. In GEOS this option is called NO_TOPO.

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_precision

Examples

```
>>> from shapely import LineString, Point
>>> set_precision(Point(0.9, 0.9), 1.0)
<POINT (1 1)>
>>> set_precision(Point(0.9, 0.9, 0.9), 1.0)
<POINT Z (1 1 0.9)>
>>> set_precision(LineString([(0, 0), (0, 0.1), (0, 1), (1, 1)]), 1.0)
<LINESTRING (0 0, 0 1, 1 1)>
>>> set_precision(LineString([(0, 0), (0, 0.1), (0.1, 0.1)]), 1.0, mode="valid_
↪output")
<LINESTRING Z EMPTY>
>>> set_precision(LineString([(0, 0), (0, 0.1), (0.1, 0.1)]), 1.0, mode="pointwise")
<LINESTRING (0 0, 0 0, 0 0)>
>>> set_precision(LineString([(0, 0), (0, 0.1), (0.1, 0.1)]), 1.0, mode="keep_
↪collapsed")
<LINESTRING (0 0, 0 0)>
>>> set_precision(None, 1.0) is None
True
```

5.7.22 shapely.force_2d

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

Forces the dimensionality of a geometry to 2D.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point, Polygon, from_wkt
>>> force_2d(Point(0, 0, 1))
<POINT (0 0)>
>>> force_2d(Point(0, 0))
<POINT (0 0)>
>>> force_2d(LineString([(0, 0, 0), (0, 1, 1), (1, 1, 2)]))
<LINESTRING (0 0, 0 1, 1 1)>
>>> force_2d(from_wkt("POLYGON Z EMPTY"))
<POLYGON EMPTY>
>>> force_2d(None) is None
True
```

5.7.23 shapely.force_3d

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

Forces the dimensionality of a geometry to 3D.

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

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

Parameters

geometry
[Geometry or array_like]

z
[float or array_like, default 0.0]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point
>>> force_3d(Point(0, 0), z=3)
<POINT Z (0 0 3)>
>>> force_3d(Point(0, 0, 0), z=3)
<POINT Z (0 0 0)>
>>> force_3d(LineString([(0, 0), (0, 1), (1, 1)]))
<LINESTRING Z (0 0 0, 0 1 0, 1 1 0)>
>>> force_3d(None) is None
True
```

5.8 Geometry creation

points(coords[, y, z, indices, out])	Create an array of points.
linestrings(coords[, y, z, indices, out])	Create an array of linestrings.
linearrings(coords[, y, z, indices, out])	Create an array of linearrings.
polygons(geometries[, holes, indices, out])	Create an array of polygons.
multipoints(geometries[, indices, out])	Create multipoints from arrays of points
multilinestrings(geometries[, indices, out])	Create multilinestrings from arrays of linestrings
multipolygons(geometries[, indices, out])	Create multipolygons from arrays of polygons
geometrycollections(geometries[, indices, out])	Create geometrycollections from arrays of geometries
box(xmin, ymin, xmax, ymax[, ccw])	Create box polygons.
prepare(geometry, **kwargs)	Prepare a geometry, improving performance of other operations.
destroy_prepared(geometry, **kwargs)	Destroy the prepared part of a geometry, freeing up memory.
empty(shape[, geom_type, order])	Create a geometry array prefilled with None or with empty geometries.

5.8.1 shapely.points

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

Create an array of points.

Parameters

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

y
[array_like, optional]

z
[array_like, optional]

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

Notes

- GEOS 3.10, 3.11 and 3.12 automatically converts POINT (nan nan) to POINT EMPTY.
- GEOS 3.10 and 3.11 will transform a 3D point to 2D if its Z coordinate is NaN.
- Usage of the `y` and `z` arguments will prevents lazy evaluation in `dask`. Instead provide the coordinates as an array with shape `(..., 2)` or `(..., 3)` using only the `coords` argument.

Examples

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

5.8.2 shapely.linestrings

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

Create an array of linestrings.

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

Parameters

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

y
[array_like, optional]

z
[array_like, optional]

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

Notes

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

Examples

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

5.8.3 shapely.linearrings

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

Create an array of linearrings.

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

Parameters

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

y
: [array_like, optional]

z
: [array_like, optional]

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

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

****kwargs**
: See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

See also:

linestrings

Notes

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

Examples

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

5.8.4 shapely.polygons

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

Create an array of polygons.

Parameters

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

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

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

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

****kwargs**
: See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

Examples

Polygons are constructed from rings:

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

Or from coordinates directly:

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

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

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

Or using the `indices` argument:

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

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

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

5.8.5 shapely.multipoints

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

Create multipoints from arrays of points

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

Examples

Multipoints are constructed from points:

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

Or from coordinates directly:

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

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

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

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

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

5.8.6 shapely.multilinestrings

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

Create multilinestrings from arrays of linestrings

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

See also:

multipoints

5.8.7 shapely.multipolygons

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

Create multipolygons from arrays of polygons

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

See also:

multipoints

5.8.8 shapely.geometrycollections

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

Create geometrycollections from arrays of geometries

Parameters

geometries
[array_like] An array of geometries

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments. Ignored if `indices` is provided.

See also:

multipoints

5.8.9 shapely.box

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

Create box polygons.

Parameters

xmin
[array_like]

ymin
[array_like]

xmax
[array_like]

ymax
[array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

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

5.8.10 shapely.prepare

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.

This function does not return any values; geometries are modified in place.

Parameters

geometry
[Geometry or array_like] Geometries are changed in place

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_prepared
Identify whether a geometry is prepared already.

destroy_prepared
Destroy the prepared part of a geometry.

Examples

```
>>> from shapely import Point, buffer, prepare, contains_properly
>>> poly = buffer(Point(1.0, 1.0), 1)
>>> prepare(poly)
>>> contains_properly(poly, [Point(0.0, 0.0), Point(0.5, 0.5)]).tolist()
[False, True]
```

5.8.11 shapely.destroy_prepared

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

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

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

Parameters

geometry
[Geometry or array_like] Geometries are changed inplace

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

prepare

5.8.12 shapely.empty

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

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

Parameters

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

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

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

Examples

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

5.9 Input/Output

from_geojson(geometry[, on_invalid])	Creates geometries from GeoJSON representations (strings).
from_ragged_array(geometry_type, coords[, ...])	Creates geometries from a contiguous array of coordinates and offset arrays.
from_wkb(geometry[, on_invalid])	Creates geometries from the Well-Known Binary (WKB) representation.
from_wkt(geometry[, on_invalid])	Creates geometries from the Well-Known Text (WKT) representation.
to_geojson(geometry[, indent])	Converts to the GeoJSON representation of a Geometry.
to_ragged_array(geometries[, include_z])	Converts geometries to a ragged array representation using a contiguous array of coordinates and offset arrays.
to_wkb(geometry[, hex, output_dimension, ...])	Converts to the Well-Known Binary (WKB) representation of a Geometry.
to_wkt(geometry[, rounding_precision, trim, ...])	Converts to the Well-Known Text (WKT) representation of a Geometry.

5.9.1 shapely.from_geojson

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

Creates geometries from GeoJSON representations (strings).

Note: 'from_geojson' requires at least GEOS 3.10.1.

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

The GeoJSON format is defined in RFC 7946.

The following are currently unsupported:

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

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_parts

Examples

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

5.9.2 shapely.from_ragged_array

`from_ragged_array`(*geometry_type*, *coords*, *offsets=None*)

Creates geometries from a contiguous array of coordinates and offset arrays.

This function creates geometries from the ragged array representation as returned by `to_ragged_array`.

This follows the in-memory layout of the variable size list arrays defined by Apache Arrow, as specified for geometries by the GeoArrow project: https://github.com/geoarrow/geoarrow.

See *to_ragged_array()* for more details.

Parameters

geometry_type
[GeometryType] The type of geometry to create.

coords
[np.ndarray] Contiguous array of shape (n, 2) or (n, 3) of all coordinates for the geometries.

offsets: tuple of np.ndarray
Offset arrays that allow to reconstruct the geometries based on the flat coordinates array. The number of offset arrays depends on the geometry type. See https://github.com/geoarrow/geoarrow/blob/main/format.md for details.

Returns

np.ndarray
Array of geometries (1-dimensional).

See also:

to_ragged_array

5.9.3 shapely.from_wkb

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

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

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

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

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

5.9.4 shapely.from_wkt

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

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

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

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from_wkt('POINT (0 0)')
<POINT (0 0)>
```

5.9.5 shapely.to_geojson

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

Converts to the GeoJSON representation of a Geometry.

Note: 'to_geojson' requires at least GEOS 3.10.0.

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

The following are currently unsupported:

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

Parameters

geometry
[str, bytes or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import Point
>>> point = Point(1, 1)
>>> to_geojson(point)
'{"type":"Point","coordinates":[1.0,1.0]}'
>>> print(to_geojson(point, indent=2))
{
  "type": "Point",
  "coordinates": [
      1.0,
      1.0
  ]
}
```

5.9.6 shapely.to_ragged_array

`to_ragged_array`(*geometries*, *include_z=None*)

Converts geometries to a ragged array representation using a contiguous array of coordinates and offset arrays.

This function converts an array of geometries to a ragged array (i.e. irregular array of arrays) of coordinates, represented in memory using a single contiguous array of the coordinates, and up to 3 offset arrays that keep track where each sub-array starts and ends.

This follows the in-memory layout of the variable size list arrays defined by Apache Arrow, as specified for geometries by the GeoArrow project: https://github.com/geoarrow/geoarrow.

Parameters

geometries
[array_like] Array of geometries (1-dimensional).

include_z
[bool, default None] If False, return 2D geometries. If True, include the third dimension in the output (if a geometry has no third dimension, the z-coordinates will be NaN). By default, will infer the dimensionality from the input geometries. Note that this inference can be unreliable with empty geometries (for a guaranteed result, it is recommended to specify the keyword).

Returns

tuple of (geometry_type, coords, offsets)

geometry_type
[GeometryType] The type of the input geometries (required information for roundtrip).

coords
[np.ndarray] Contiguous array of shape (n, 2) or (n, 3) of all coordinates of all input geometries.

offsets: tuple of np.ndarray
Offset arrays that make it possible to reconstruct the geometries from the flat coordinates array. The number of offset arrays depends on the geometry type. See https://github.com/geoarrow/geoarrow/blob/main/format.md for details.

See also:

from_ragged_array

Notes

Mixed singular and multi geometry types of the same basic type are allowed (e.g., Point and MultiPoint) and all singular types will be treated as multi types. GeometryCollections and other mixed geometry types are not supported.

Examples

Consider a Polygon with one hole (interior ring):

```
>>> import shapely
>>> polygon = shapely.Polygon(
...     [(0, 0), (10, 0), (10, 10), (0, 10)],
...     holes=[[(2, 2), (3, 2), (2, 3)]]
... )
>>> polygon
<POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 3 2, 2 3, 2 2))>
```

This polygon can be thought of as a list of rings (first ring is the exterior ring, subsequent rings are the interior rings), and each ring as a list of coordinate pairs. This is very similar to how GeoJSON represents the coordinates:

```
>>> import json
>>> json.loads(shapely.to_geojson(polygon))["coordinates"]
[[[0.0, 0.0], [10.0, 0.0], [10.0, 10.0], [0.0, 10.0], [0.0, 0.0]],
 [[2.0, 2.0], [3.0, 2.0], [2.0, 3.0], [2.0, 2.0]]]
```

This function will return a similar list of lists of lists, but using a single contiguous array of coordinates, and multiple arrays of offsets:

```
>>> geometry_type, coords, offsets = shapely.to_ragged_array([polygon])
>>> geometry_type
<GeometryType.POLYGON: 3>
>>> coords
array([[ 0.,  0.],
       [10.,  0.],
       [10., 10.],
       [ 0., 10.],
       [ 0.,  0.],
       [ 2.,  2.],
       [ 3.,  2.],
       [ 2.,  3.],
       [ 2.,  2.]])
```

```
>>> offsets
(array([0, 5, 9]), array([0, 2]))
```

As an example how to interpret the offsets: the i-th ring in the coordinates is represented by `offsets[0][i]` to `offsets[0][i+1]`:

```
>>> exterior_ring_start, exterior_ring_end = offsets[0][0], offsets[0][1]
>>> coords[exterior_ring_start:exterior_ring_end]
array([[ 0.,  0.],
       [10.,  0.],
       [10., 10.],
       [ 0., 10.],
       [ 0.,  0.]])
```

5.9.7 shapely.to_wkb

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

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

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

The following limitations apply to WKB serialization:

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

Parameters

geometry
[Geometry or array_like]

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

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

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

include_srid
[bool, default False] If True, the SRID is be included in WKB (this is an extension to the OGC WKB specification). Not allowed when flavor is "iso".

flavor
[{"iso", "extended"}, default "extended"] Which flavor of WKB will be returned. The flavor determines how extra dimensionality is encoded with the type number, and whether SRID can be included in the WKB. ISO flavor is "more standard" for 3D output, and does not support SRID embedding. Both flavors are equivalent when `output_dimension=2` (or with 2D geometries) and `include_srid=False`. The *from_wkb* function can read both flavors.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import Point
>>> point = Point(1, 1)
>>> to_wkb(point, byte_order=1)
b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\xf0?'
>>> to_wkb(point, hex=True, byte_order=1)
'0101000000000000000000F03F000000000000F03F'
```

5.9.8 shapely.to_wkt

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

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

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

The following limitations apply to WKT serialization:

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

Parameters

geometry
[Geometry or array_like]

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

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

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

```
>>> from shapely import Point
>>> to_wkt(Point(0, 0))
'POINT (0 0)'
>>> to_wkt(Point(0, 0), rounding_precision=3, trim=False)
'POINT (0.000 0.000)'
>>> to_wkt(Point(0, 0), rounding_precision=-1, trim=False)
'POINT (0.0000000000000000 0.0000000000000000)'
>>> to_wkt(Point(1, 2, 3), trim=True)
'POINT Z (1 2 3)'
>>> to_wkt(Point(1, 2, 3), trim=True, output_dimension=2)
'POINT (1 2)'
>>> to_wkt(Point(1, 2, 3), trim=True, old_3d=True)
'POINT (1 2 3)'
```

5.10 Measurement

area(geometry, **kwargs)	Computes the area of a (multi)polygon.
distance(a, b, **kwargs)	Computes the Cartesian distance between two geometries.
bounds(geometry, **kwargs)	Computes the bounds (extent) of a geometry.
total_bounds(geometry, **kwargs)	Computes the total bounds (extent) of the geometry.
length(geometry, **kwargs)	Computes the length of a (multi)linestring or polygon perimeter.
hausdorff_distance(a, b[, densify])	Compute the discrete Hausdorff distance between two geometries.
frechet_distance(a, b[, densify])	Compute the discrete Fréchet distance between two geometries.
minimum_clearance(geometry, **kwargs)	Computes the Minimum Clearance distance.
minimum_bounding_radius(geometry, **kwargs)	Computes the radius of the minimum bounding circle that encloses an input geometry.

5.10.1 shapely.area

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

Computes the area of a (multi)polygon.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import MultiPolygon, Polygon
>>> polygon = Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)])
>>> area(polygon)
100.0
>>> area(MultiPolygon([polygon, Polygon([(10, 10), (10, 20), (20, 20), (20, 10),␣
↪(10, 10)])]))
200.0
>>> area(Polygon())
0.0
>>> area(None)
nan
```

5.10.2 shapely.distance

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

Computes the Cartesian distance between two geometries.

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> point = Point(0, 0)
>>> distance(Point(10, 0), point)
10.0
>>> distance(LineString([(1, 1), (1, -1)]), point)
1.0
>>> distance(Polygon([(3, 0), (5, 0), (5, 5), (3, 5), (3, 0)]), point)
3.0
>>> distance(Point(), point)
nan
```

(continues on next page)

(continued from previous page)

```
>>> distance(None, point)
nan
```

5.10.3 shapely.bounds

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

Computes the bounds (extent) of a geometry.

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> bounds(Point(2, 3)).tolist()
[2.0, 3.0, 2.0, 3.0]
>>> bounds(LineString([(0, 0), (0, 2), (3, 2)])).tolist()
[0.0, 0.0, 3.0, 2.0]
>>> bounds(Polygon()).tolist()
[nan, nan, nan, nan]
>>> bounds(None).tolist()
[nan, nan, nan, nan]
```

5.10.4 shapely.total_bounds

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

Computes the total bounds (extent) of the geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Returns

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

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> total_bounds(Point(2, 3)).tolist()
[2.0, 3.0, 2.0, 3.0]
>>> total_bounds([Point(2, 3), Point(4, 5)]).tolist()
[2.0, 3.0, 4.0, 5.0]
>>> total_bounds([
...     LineString([(0, 1), (0, 2), (3, 2)]),
...     LineString([(4, 4), (4, 6), (6, 7)])
... ]).tolist()
[0.0, 1.0, 6.0, 7.0]
>>> total_bounds(Polygon()).tolist()
[nan, nan, nan, nan]
>>> total_bounds([Polygon(), Point(2, 3)]).tolist()
[2.0, 3.0, 2.0, 3.0]
>>> total_bounds(None).tolist()
[nan, nan, nan, nan]
```

5.10.5 shapely.length

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

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, MultiLineString, Polygon
>>> length(LineString([(0, 0), (0, 2), (3, 2)]))
5.0
>>> length(MultiLineString([
...     LineString([(0, 0), (1, 0)]),
...     LineString([(1, 0), (2, 0)])
... ]))
2.0
>>> length(Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)]))
40.0
>>> length(LineString())
0.0
>>> length(None)
nan
```

5.10.6 shapely.hausdorff_distance

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

Compute the discrete Hausdorff distance between two geometries.

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

Parameters

a, b
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString
>>> line1 = LineString([(130, 0), (0, 0), (0, 150)])
>>> line2 = LineString([(10, 10), (10, 150), (130, 10)])
>>> hausdorff_distance(line1, line2)
14.14...
>>> hausdorff_distance(line1, line2, densify=0.5)
70.0
>>> hausdorff_distance(line1, LineString())
nan
>>> hausdorff_distance(line1, None)
nan
```

5.10.7 shapely.frechet_distance

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

Compute the discrete Fréchet distance between two geometries.

Note: 'frechet_distance' requires at least GEOS 3.7.0.

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

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

Parameters

a, b
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString
>>> line1 = LineString([(0, 0), (100, 0)])
>>> line2 = LineString([(0, 0), (50, 50), (100, 0)])
>>> frechet_distance(line1, line2)
70.71...
>>> frechet_distance(line1, line2, densify=0.5)
50.0
>>> frechet_distance(line1, LineString())
nan
>>> frechet_distance(line1, None)
nan
```

5.10.8 shapely.minimum_clearance

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

Computes the Minimum Clearance distance.

Note: 'minimum_clearance' requires at least GEOS 3.6.0.

A geometry's "minimum clearance" is the smallest distance by which a vertex of the geometry could be moved to produce an invalid geometry.

If no minimum clearance exists for a geometry (for example, a single point, or an empty geometry), infinity is returned.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import Polygon
>>> polygon = Polygon([(0, 0), (0, 10), (5, 6), (10, 10), (10, 0), (5, 4), (0, 0)])
>>> minimum_clearance(polygon)
2.0
>>> minimum_clearance(Polygon())
inf
>>> minimum_clearance(None)
nan
```

5.10.9 shapely.minimum_bounding_radius

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

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

Note: 'minimum_bounding_radius' requires at least GEOS 3.8.0.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

minimum_bounding_circle

Examples

```
>>> from shapely import GeometryCollection, LineString, MultiPoint, Point, Polygon
>>> minimum_bounding_radius(Polygon([(0, 5), (5, 10), (10, 5), (5, 0), (0, 5)]))
5.0
>>> minimum_bounding_radius(LineString([(1, 1), (1, 10)]))
4.5
>>> minimum_bounding_radius(MultiPoint([(2, 2), (4, 2)]))
1.0
>>> minimum_bounding_radius(Point(0, 1))
0.0
>>> minimum_bounding_radius(GeometryCollection())
0.0
```

5.11 Predicates

has_z(geometry, **kwargs)	Returns True if a geometry has a Z coordinate.
is_ccw(geometry, **kwargs)	Returns True if a linestring or linearring is counterclockwise.
is_closed(geometry, **kwargs)	Returns True if a linestring's first and last points are equal.
is_empty(geometry, **kwargs)	Returns True if a geometry is an empty point, polygon, etc.
is_geometry(geometry, **kwargs)	Returns True if the object is a geometry
is_missing(geometry, **kwargs)	Returns True if the object is not a geometry (None)
is_prepared(geometry, **kwargs)	Returns True if a Geometry is prepared.
is_ring(geometry, **kwargs)	Returns True if a linestring is closed and simple.
is_simple(geometry, **kwargs)	Returns True if a Geometry has no anomalous geometric points, such as self-intersections or self tangency.
is_valid(geometry, **kwargs)	Returns True if a geometry is well formed.
is_valid_input(geometry, **kwargs)	Returns True if the object is a geometry or None
is_valid_reason(geometry, **kwargs)	Returns a string stating if a geometry is valid and if not, why.
crosses(a, b, **kwargs)	Returns True if A and B spatially cross.
contains(a, b, **kwargs)	Returns True if geometry B is completely inside geometry A.
contains_xy(geom, x[, y])	Returns True if the Point (x, y) is completely inside geometry A.
contains_properly(a, b, **kwargs)	Returns True if geometry B is completely inside geometry A, with no common boundary points.
covered_by(a, b, **kwargs)	Returns True if no point in geometry A is outside geometry B.
covers(a, b, **kwargs)	Returns True if no point in geometry B is outside geometry A.
disjoint(a, b, **kwargs)	Returns True if A and B do not share any point in space.
dwithin(a, b, distance, **kwargs)	Returns True if the geometries are within a given distance.
equals(a, b, **kwargs)	Returns True if A and B are spatially equal.
intersects(a, b, **kwargs)	Returns True if A and B share any portion of space.
intersects_xy(geom, x[, y])	Returns True if A and the Point (x, y) share any portion of space.
overlaps(a, b, **kwargs)	Returns True if A and B spatially overlap.
touches(a, b, **kwargs)	Returns True if the only points shared between A and B are on the boundary of A and B.
within(a, b, **kwargs)	Returns True if geometry A is completely inside geometry B.
equals_exact(a, b[, tolerance])	Returns True if A and B are structurally equal.
relate(a, b, **kwargs)	Returns a string representation of the DE-9IM intersection matrix.
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.

5.11.1 shapely.has_z

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

Returns True if a geometry has a Z coordinate.

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

get_coordinate_dimension

Examples

```
>>> from shapely import Point
>>> has_z(Point(0, 0))
False
>>> has_z(Point(0, 0, 0))
True
>>> has_z(Point(0, 0, float("nan")))
False
```

5.11.2 shapely.is_ccw

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

Returns True if a linestring or linearring is counterclockwise.

Note: 'is_ccw' requires at least GEOS 3.7.0.

Note that there are no checks on whether lines are actually closed and not self-intersecting, while this is a requirement for is_ccw. The recommended usage of this function for linestrings is `is_ccw(g) & is_simple(g)` and for linearrings `is_ccw(g) & is_valid(g)`.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_simple
Checks if a linestring is closed and simple.

is_valid
Checks additionally if the geometry is simple.

Examples

```
>>> from shapely import LinearRing, LineString, Point
>>> is_ccw(LinearRing([(0, 0), (0, 1), (1, 1), (0, 0)]))
False
>>> is_ccw(LinearRing([(0, 0), (1, 1), (0, 1), (0, 0)]))
True
>>> is_ccw(LineString([(0, 0), (1, 1), (0, 1)]))
False
>>> is_ccw(Point(0, 0))
False
```

5.11.3 shapely.is_closed

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

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

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_ring
Checks additionally if the geometry is simple.

Examples

```
>>> from shapely import LineString, Point
>>> is_closed(LineString([(0, 0), (1, 1)]))
False
>>> is_closed(LineString([(0, 0), (0, 1), (1, 1), (0, 0)]))
True
>>> is_closed(Point(0, 0))
False
```

5.11.4 shapely.is_empty

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

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

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_missing
checks if the object is a geometry

Examples

```
>>> from shapely import Point
>>> is_empty(Point())
True
>>> is_empty(Point(0, 0))
False
>>> is_empty(None)
False
```

5.11.5 shapely.is_geometry

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

Returns True if the object is a geometry

Parameters

geometry
[any object or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_missing
check if an object is missing (None)

is_valid_input
check if an object is a geometry or None

Examples

```
>>> from shapely import GeometryCollection, Point
>>> is_geometry(Point(0, 0))
True
>>> is_geometry(GeometryCollection())
True
>>> is_geometry(None)
False
>>> is_geometry("text")
False
```

5.11.6 shapely.is_missing

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

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

Parameters

geometry
[any object or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

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

```
>>> from shapely import GeometryCollection, Point
>>> is_missing(Point(0, 0))
False
>>> is_missing(GeometryCollection())
False
>>> is_missing(None)
True
>>> is_missing("text")
False
```

5.11.7 shapely.is_prepared

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

Returns True if a Geometry is prepared.

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

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_valid_input
check if an object is a geometry or None

prepare
prepare a geometry

Examples

```
>>> from shapely import Point, prepare
>>> geometry = Point(0, 0)
>>> is_prepared(Point(0, 0))
False
>>> prepare(geometry)
>>> is_prepared(geometry)
True
>>> is_prepared(None)
False
```

5.11.8 shapely.is_ring

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

Returns True if a linestring is closed and simple.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_closed
Checks only if the geometry is closed.

is_simple
Checks only if the geometry is simple.

Examples

```
>>> from shapely import LineString, Point
>>> is_ring(Point(0, 0))
False
>>> geom = LineString([(0, 0), (1, 1)])
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(False, True, False)
>>> geom = LineString([(0, 0), (0, 1), (1, 1), (0, 0)])
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(True, True, True)
>>> geom = LineString([(0, 0), (1, 1), (0, 1), (1, 0), (0, 0)])
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(True, False, False)
```

5.11.9 shapely.is_simple

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

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

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

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_ring
Checks additionally if the geometry is closed.

is_valid
Checks whether a geometry is well formed.

Examples

```
>>> from shapely import LineString, Polygon
>>> is_simple(Polygon([(1, 1), (2, 1), (2, 2), (1, 1)]))
True
>>> is_simple(LineString([(0, 0), (1, 1), (0, 1), (1, 0), (0, 0)]))
False
>>> is_simple(None)
False
```

5.11.10 shapely.is_valid

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

Returns True if a geometry is well formed.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_valid_reason
Returns the reason in case of invalid.

Examples

```
>>> from shapely import GeometryCollection, LineString, Polygon
>>> is_valid(LineString([(0, 0), (1, 1)]))
True
>>> is_valid(Polygon([(0, 0), (1, 1), (1, 2), (1, 1), (0, 0)]))
False
>>> is_valid(GeometryCollection())
True
>>> is_valid(None)
False
```

5.11.11 shapely.is_valid_input

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

Returns True if the object is a geometry or None

Parameters

geometry
[any object or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_geometry
checks if an object is a geometry

is_missing
checks if an object is None

Examples

```
>>> from shapely import GeometryCollection, Point
>>> is_valid_input(Point(0, 0))
True
>>> is_valid_input(GeometryCollection())
True
>>> is_valid_input(None)
True
>>> is_valid_input(1.0)
False
>>> is_valid_input("text")
False
```

5.11.12 shapely.is_valid_reason

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

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

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_valid
returns True or False

Examples

```
>>> from shapely import LineString, Polygon
>>> is_valid_reason(LineString([(0, 0), (1, 1)]))
'Valid Geometry'
>>> is_valid_reason(Polygon([(0, 0), (1, 1), (1, 2), (1, 1), (0, 0)]))
'Ring Self-intersection[1 1]'
>>> is_valid_reason(None) is None
True
```

5.11.13 shapely.crosses

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

Returns True if A and B spatially cross.

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import LineString, MultiPoint, Point, Polygon
>>> line = LineString([(0, 0), (1, 1)])
>>> # A contains B:
>>> crosses(line, Point(0.5, 0.5))
False
>>> # A and B intersect at a point but do not share all points:
>>> crosses(line, MultiPoint([(0, 1), (0.5, 0.5)]))
True
>>> crosses(line, LineString([(0, 1), (1, 0)]))
True
>>> # A is contained by B; their intersection is a line (same dimension):
>>> crosses(line, LineString([(0, 0), (2, 2)]))
False
>>> area = 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, LineString([(0, 0), (2, 2)]))
True
>>> # A contains B:
>>> crosses(area, Point(0.5, 0.5))
False
>>> # A contains some but not all points of B; they intersect at a point:
>>> crosses(area, MultiPoint([(2, 2), (0.5, 0.5)]))
True
```

5.11.14 shapely.contains

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

Returns True if geometry B is completely inside geometry A.

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

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)

contains_xy
variant for checking against a Point with x, y coordinates

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> line = LineString([(0, 0), (1, 1)])
>>> contains(line, Point(0, 0))
False
>>> contains(line, Point(0.5, 0.5))
True
>>> area = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> contains(area, Point(0, 0))
False
>>> contains(area, line)
True
>>> contains(area, LineString([(0, 0), (2, 2)]))
False
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> contains(polygon_with_hole, Point(1, 1))
True
>>> contains(polygon_with_hole, Point(2, 2))
False
>>> contains(polygon_with_hole, LineString([(1, 1), (5, 5)]))
False
>>> contains(area, area)
True
>>> contains(area, None)
False
```

5.11.15 shapely.contains_xy

contains_xy(*geom*, *x*, *y=None*, ***kwargs*)

Returns True if the Point (x, y) is completely inside geometry A.

This is a special-case (and faster) variant of the *contains* function which avoids having to create a Point object if you start from x/y coordinates.

Note that in the case of points, the *contains_properly* predicate is equivalent to *contains*.

See the docstring of *contains* for more details about the predicate.

Parameters

geom
[Geometry or array_like]

x, y
[float or array_like] Coordinates as separate x and y arrays, or a single array of coordinate x, y tuples.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

contains
variant taking two geometries as input

Notes

If you compare a small number of polygons or lines with many points, it can be beneficial to prepare the geometries in advance using *shapely.prepare()*.

Examples

```
>>> from shapely import Point, Polygon
>>> area = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> contains(area, Point(0.5, 0.5))
True
>>> contains_xy(area, 0.5, 0.5)
True
```

5.11.16 shapely.contains_properly

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

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

A contains B properly if B intersects the interior of A but not the boundary (or exterior). This means that a geometry A does not "contain properly" itself, which contrasts with the `contains` function, where common points on the boundary are allowed.

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

contains
contains which allows common boundary points

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import Polygon
>>> area1 = Polygon([(0, 0), (3, 0), (3, 3), (0, 3), (0, 0)])
>>> area2 = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> area3 = 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
```

5.11.17 shapely.covered_by

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> line = LineString([(0, 0), (1, 1)])
>>> covered_by(Point(0, 0), line)
True
>>> covered_by(Point(0.5, 0.5), line)
True
>>> area = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> covered_by(Point(0, 0), area)
True
>>> covered_by(line, area)
True
>>> covered_by(LineString([(0, 0), (2, 2)]), area)
False
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> covered_by(Point(1, 1), polygon_with_hole)
True
>>> covered_by(Point(2, 2), polygon_with_hole)
True
>>> covered_by(LineString([(1, 1), (5, 5)]), polygon_with_hole)
False
>>> covered_by(area, area)
True
>>> covered_by(None, area)
False
```

5.11.18 shapely.covers

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

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

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> line = LineString([(0, 0), (1, 1)])
>>> covers(line, Point(0, 0))
True
>>> covers(line, Point(0.5, 0.5))
True
>>> area = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> covers(area, Point(0, 0))
True
>>> covers(area, line)
True
>>> covers(area, LineString([(0, 0), (2, 2)]))
False
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> covers(polygon_with_hole, Point(1, 1))
True
>>> covers(polygon_with_hole, Point(2, 2))
True
>>> covers(polygon_with_hole, LineString([(1, 1), (5, 5)]))
False
>>> covers(area, area)
True
>>> covers(area, None)
False
```

5.11.19 shapely.disjoint

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

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import GeometryCollection, LineString, Point
>>> line = LineString([(0, 0), (1, 1)])
>>> disjoint(line, Point(0, 0))
False
>>> disjoint(line, Point(0, 1))
True
>>> disjoint(line, LineString([(0, 2), (2, 0)]))
False
>>> empty = GeometryCollection()
>>> disjoint(line, empty)
True
>>> disjoint(empty, empty)
True
>>> disjoint(empty, None)
False
>>> disjoint(None, None)
False
```

5.11.20 shapely.dwithin

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

Returns True if the geometries are within a given distance.

Note: 'dwithin' requires at least GEOS 3.10.0.

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

Parameters

a, b
[Geometry or array_like]

distance
[float] Negative distances always return False.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

distance
compute the actual distance between A and B

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import Point
>>> point = Point(0.5, 0.5)
>>> dwithin(point, Point(2, 0.5), 2)
True
>>> dwithin(point, Point(2, 0.5), [2, 1.5, 1]).tolist()
[True, True, False]
>>> dwithin(point, Point(0.5, 0.5), 0)
True
>>> dwithin(point, None, 100)
False
```

5.11.21 shapely.equals

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

Returns True if A and B are spatially equal.

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

Examples

```
>>> from shapely import GeometryCollection, LineString, Polygon
>>> line = LineString([(0, 0), (5, 5), (10, 10)])
>>> equals(line, LineString([(0, 0), (10, 10)]))
True
>>> equals(Polygon(), GeometryCollection())
True
>>> equals(None, None)
False
```

5.11.22 shapely.intersects

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

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

Intersects implies that overlaps, touches and within are True.

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

prepare
improve performance by preparing a (the first argument)

intersects_xy
variant for checking against a Point with x, y coordinates

Examples

```
>>> from shapely import LineString, Point
>>> line = LineString([(0, 0), (1, 1)])
>>> intersects(line, Point(0, 0))
True
>>> intersects(line, Point(0, 1))
False
>>> intersects(line, LineString([(0, 2), (2, 0)]))
True
>>> intersects(None, None)
False
```

5.11.23 shapely.intersects_xy

intersects_xy(*geom*, *x*, *y=None*, ***kwargs*)

Returns True if A and the Point (x, y) share any portion of space.

This is a special-case (and faster) variant of the *intersects* function which avoids having to create a Point object if you start from x/y coordinates.

See the docstring of *intersects* for more details about the predicate.

Parameters

geom
[Geometry or array_like]

x, y
[float or array_like] Coordinates as separate x and y arrays, or a single array of coordinate x, y tuples.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

intersects
variant taking two geometries as input

Notes

If you compare a single or few geometries with many points, it can be beneficial to prepare the geometries in advance using *shapely.prepare()*.

The *touches* predicate can be determined with this function by getting the boundary of the geometries: `intersects_xy(boundary(geom), x, y)`.

Examples

```
>>> from shapely import LineString, Point
>>> line = LineString([(0, 0), (1, 1)])
>>> intersects(line, Point(0, 0))
True
>>> intersects_xy(line, 0, 0)
True
```

5.11.24 shapely.overlaps

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

Returns True if A and B spatially overlap.

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> poly = 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, Polygon([(0, 0), (0, 2), (2, 2), (2, 0), (0, 0)]))
False
>>> # A partially overlaps with B:
>>> overlaps(poly, Polygon([(2, 2), (2, 6), (6, 6), (6, 2), (2, 2)]))
True
>>> line = LineString([(2, 2), (6, 6)])
>>> # A and B are different dimensions; they cannot overlap:
>>> overlaps(poly, line)
False
>>> overlaps(poly, Point(2, 2))
False
>>> # A and B share some but not all points:
>>> overlaps(line, LineString([(0, 0), (4, 4)]))
True
>>> # A and B intersect only at a point (lower dimension); they do not overlap
>>> overlaps(line, LineString([(6, 0), (0, 6)]))
False
>>> overlaps(poly, None)
False
>>> overlaps(None, None)
False
```

5.11.25 shapely.touches

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> line = LineString([(0, 2), (2, 0)])
>>> touches(line, Point(0, 2))
True
>>> touches(line, Point(1, 1))
False
>>> touches(line, LineString([(0, 0), (1, 1)]))
True
>>> touches(line, LineString([(0, 0), (2, 2)]))
False
>>> area = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> touches(area, Point(0.5, 0))
True
>>> touches(area, Point(0.5, 0.5))
False
>>> touches(area, line)
True
>>> touches(area, Polygon([(0, 1), (1, 1), (1, 2), (0, 2), (0, 1)]))
True
```

5.11.26 shapely.within

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

Returns True if geometry A is completely inside geometry B.

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

prepare
improve performance by preparing a (the first argument)

Examples

```
>>> from shapely import LineString, Point, Polygon
>>> line = LineString([(0, 0), (1, 1)])
>>> within(Point(0, 0), line)
False
>>> within(Point(0.5, 0.5), line)
True
>>> area = Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> within(Point(0, 0), area)
False
>>> within(line, area)
True
>>> within(LineString([(0, 0), (2, 2)]), area)
False
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> within(Point(1, 1), polygon_with_hole)
True
>>> within(Point(2, 2), polygon_with_hole)
False
>>> within(LineString([(1, 1), (5, 5)]), polygon_with_hole)
False
>>> within(area, area)
True
>>> within(None, area)
False
```

5.11.27 shapely.equals_exact

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

Returns True if A and B are structurally equal.

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

Parameters

a, b
[Geometry or array_like]

tolerance
[float or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

equals
Check if A and B are spatially equal.

Examples

```
>>> from shapely import Point, Polygon
>>> point1 = Point(50, 50)
>>> point2 = 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 = Polygon([(0, 0), (1, 1), (0, 1), (0, 0)])
>>> polygon2 = Polygon([(0, 0), (0, 1), (1, 1), (0, 0)])
>>> equals_exact(polygon1, polygon2)
False
>>> equals(polygon1, polygon2)
True
```

5.11.28 shapely.relate

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

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

Parameters

a, b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point
>>> point = Point(0, 0)
>>> line = LineString([(0, 0), (1, 1)])
>>> relate(point, line)
'F0FFFF102'
```

5.11.29 shapely.relate_pattern

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

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

This function compares the DE-9IM code string for two geometries against a specified pattern. If the string matches the pattern then `True` is returned, otherwise `False`. The pattern specified can be an exact match (`0`, `1` or `2`), a boolean match (uppercase `T` or `F`), or a wildcard (`*`). For example, the pattern for the `within` predicate is `'T*F**F***'`.

Parameters

a, b
[Geometry or array_like]

pattern
[string]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import Point, Polygon
>>> point = Point(0.5, 0.5)
>>> square = Polygon([(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)])
>>> relate(point, square)
'0FFFFF212'
>>> relate_pattern(point, square, "T*F**F***")
True
```

5.12 Set operations

difference(a, b[, grid_size])	Returns the part of geometry A that does not intersect with geometry B.
intersection(a, b[, grid_size])	Returns the geometry that is shared between input geometries.
intersection_all(geometries[, axis])	Returns the intersection of multiple geometries.
symmetric_difference(a, b[, grid_size])	Returns the geometry that represents the portions of input geometries that do not intersect.
symmetric_difference_all(geometries[, axis])	Returns the symmetric difference of multiple geometries.
unary_union(geometries[, grid_size, axis])	Returns the union of multiple geometries.
union(a, b[, grid_size])	Merges geometries into one.
union_all(geometries[, grid_size, axis])	Returns the union of multiple geometries.
coverage_union(a, b, **kwargs)	Merges multiple polygons into one.
coverage_union_all(geometries[, axis])	Returns the union of multiple polygons of a geometry collection.

5.12.1 shapely.difference

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

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

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

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

set_precision

Examples

```
>>> from shapely import box, LineString, normalize, Polygon
>>> line = LineString([(0, 0), (2, 2)])
>>> difference(line, LineString([(1, 1), (3, 3)]))
<LINESTRING (0 0, 1 1)>
>>> difference(line, LineString())
<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))
<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)
<POLYGON ((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0))>
```

5.12.2 shapely.intersection

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

Returns the geometry that is shared between input geometries.

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

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

intersection_all
set_precision

Examples

```
>>> from shapely import box, LineString, normalize, Polygon
>>> line = LineString([(0, 0), (2, 2)])
>>> intersection(line, LineString([(1, 1), (3, 3)]))
<LINESTRING (1 1, 2 2)>
>>> box1 = box(0, 0, 2, 2)
>>> box2 = box(1, 1, 3, 3)
>>> normalize(intersection(box1, box2))
<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)
<POLYGON ((2 2, 2 1, 1 1, 1 2, 2 2))>
```

5.12.3 shapely.intersection_all

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, an empty GeometryCollection is returned.

Parameters

geometries
[array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

intersection

Examples

```
>>> from shapely import LineString
>>> line1 = LineString([(0, 0), (2, 2)])
>>> line2 = LineString([(1, 1), (3, 3)])
>>> intersection_all([line1, line2])
<LINESTRING (1 1, 2 2)>
>>> intersection_all([[line1, line2, None]], axis=1).tolist()
[<LINESTRING (1 1, 2 2)>]
>>> intersection_all([line1, None])
<LINESTRING (0 0, 2 2)>
```

5.12.4 shapely.symmetric_difference

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

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

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

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

symmetric_difference_all
set_precision

Examples

```
>>> from shapely import box, LineString, normalize
>>> line = LineString([(0, 0), (2, 2)])
>>> symmetric_difference(line, LineString([(1, 1), (3, 3)]))
<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))
<MULTIPOLYGON (((1 2, 1 3, 3 3, 3 1, 2 1, 2 2, 1 2)), ((0 0, 0 2, 1 2, 1 1, ...>
>>> box1 = box(0.1, 0.2, 2.1, 2.1)
>>> symmetric_difference(box1, box2, grid_size=1)
<MULTIPOLYGON (((2 0, 0 0, 0 2, 1 2, 1 1, 2 1, 2 0)), ((2 2, 1 2, 1 3, 3 3, ...>
```

5.12.5 shapely.symmetric_difference_all

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 an empty GeometryCollection is returned.

Parameters

geometries
[array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

symmetric_difference

Examples

```
>>> from shapely import LineString
>>> line1 = LineString([(0, 0), (2, 2)])
>>> line2 = LineString([(1, 1), (3, 3)])
>>> symmetric_difference_all([line1, line2])
<MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>
>>> symmetric_difference_all([[line1, line2, None]], axis=1).tolist()
[<MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>]
>>> symmetric_difference_all([line1, None])
<LINESTRING (0 0, 2 2)>
>>> symmetric_difference_all([None, None])
<GEOMETRYCOLLECTION EMPTY>
```

5.12.6 shapely.unary_union

unary_union(*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 an empty GeometryCollection 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.

unary_union is an alias of *union_all*.

Parameters

geometries
[array_like]

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

union
set_precision

Examples

```
>>> from shapely import box, LineString, normalize, Point
>>> line1 = LineString([(0, 0), (2, 2)])
>>> line2 = LineString([(2, 2), (3, 3)])
>>> union_all([line1, line2])
<MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>
>>> union_all([[line1, line2, None]], axis=1).tolist()
[<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]))
<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)
<POLYGON ((2 0, 0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0))>
>>> union_all([None, Point(0, 1)])
<POINT (0 1)>
>>> union_all([None, None])
<GEOMETRYCOLLECTION EMPTY>
>>> union_all([])
<GEOMETRYCOLLECTION EMPTY>
```

5.12.7 shapely.union

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

Merges geometries into one.

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

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

union_all
set_precision

Examples

```
>>> from shapely import box, LineString, normalize
>>> line = LineString([(0, 0), (2, 2)])
>>> union(line, LineString([(2, 2), (3, 3)]))
<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))
<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)
<POLYGON ((2 0, 0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0))>
```

5.12.8 shapely.union_all

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 an empty GeometryCollection 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.

unary_union is an alias of *union_all*.

Parameters

geometries
[array_like]

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

union
set_precision

Examples

```
>>> from shapely import box, LineString, normalize, Point
>>> line1 = LineString([(0, 0), (2, 2)])
>>> line2 = LineString([(2, 2), (3, 3)])
>>> union_all([line1, line2])
<MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>
>>> union_all([[line1, line2, None]], axis=1).tolist()
[<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]))
<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)
<POLYGON ((2 0, 0 0, 0 2, 1 2, 1 3, 3 3, 3 1, 2 1, 2 0))>
>>> union_all([None, Point(0, 1)])
<POINT (0 1)>
>>> union_all([None, None])
<GEOMETRYCOLLECTION EMPTY>
>>> union_all([])
<GEOMETRYCOLLECTION EMPTY>
```

5.12.9 shapely.coverage_union

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

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

Note: 'coverage_union' requires at least GEOS 3.8.0.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

coverage_union_all

Examples

```
>>> from shapely import normalize, Polygon
>>> polygon = Polygon([(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)])
>>> normalize(coverage_union(polygon, Polygon([(1, 0), (1, 1), (2, 1), (2, 0), (1,␣
↪0)])))
<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)) <POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>

5.12.10 shapely.coverage_union_all

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

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

Note: 'coverage_union_all' requires at least GEOS 3.8.0.

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

Parameters

geometries
[array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

coverage_union

Examples

```
>>> from shapely import normalize, Polygon
>>> polygon_1 = Polygon([(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)])
>>> polygon_2 = Polygon([(1, 0), (1, 1), (2, 1), (2, 0), (1, 0)])
>>> normalize(coverage_union_all([polygon_1, polygon_2]))
<POLYGON ((0 0, 0 1, 1 1, 2 1, 2 0, 1 0, 0 0))>
>>> normalize(coverage_union_all([polygon_1, None]))
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
>>> normalize(coverage_union_all([None, None]))
<MULTIPOLYGON EMPTY>
```

5.13 Constructive operations

BufferCapStyle(value)	An enumeration.
BufferJoinStyle(value)	An enumeration.
boundary(geometry, **kwargs)	Returns the topological boundary of a geometry.
buffer(geometry, distance[, quad_segs, ...])	Computes the buffer of a geometry for positive and negative buffer distance.
offset_curve(geometry, distance[, ...])	Returns a (Multi)LineString at a distance from the object on its right or its left side.
centroid(geometry, **kwargs)	Computes the geometric center (center-of-mass) of a geometry.
clip_by_rect(geometry, xmin, ymin, xmax, ...)	Returns the portion of a geometry within a rectangle.
concave_hull(geometry[, ratio, allow_holes])	Computes a concave geometry that encloses an input geometry.
convex_hull(geometry, **kwargs)	Computes the minimum convex geometry that encloses an input geometry.
delaunay_triangles(geometry[, tolerance, ...])	Computes a Delaunay triangulation around the vertices of an input geometry.
segmentize(geometry, max_segment_length, ...)	Adds vertices to line segments based on maximum segment length.
envelope(geometry, **kwargs)	Computes the minimum bounding box that encloses an input geometry.
extract_unique_points(geometry, **kwargs)	Returns all distinct vertices of an input geometry as a multipoint.
build_area(geometry, **kwargs)	Creates an areal geometry formed by the constituent linework of given geometry.
make_valid(geometry, **kwargs)	Repairs invalid geometries.
normalize(geometry, **kwargs)	Converts Geometry to normal form (or canonical form).
node(geometry, **kwargs)	Returns the fully noded version of the linear input as MultiLineString.
point_on_surface(geometry, **kwargs)	Returns a point that intersects an input geometry.
polygonize(geometries, **kwargs)	Creates polygons formed from the linework of a set of Geometries.
polygonize_full(geometries, **kwargs)	Creates polygons formed from the linework of a set of Geometries and return all extra outputs as well.
remove_repeated_points(geometry[, tolerance])	Returns a copy of a Geometry with repeated points removed.
reverse(geometry, **kwargs)	Returns a copy of a Geometry with the order of coordinates reversed.
simplify(geometry, tolerance[, ...])	Returns a simplified version of an input geometry using the Douglas-Peucker algorithm.
snap(geometry, reference, tolerance, **kwargs)	Snaps an input geometry to reference geometry's vertices.
voronoi_polygons(geometry[, tolerance, ...])	Computes a Voronoi diagram from the vertices of an input geometry.
oriented_envelope(geometry, **kwargs)	Computes the oriented envelope (minimum rotated rectangle) that encloses an input geometry, such that the resulting rectangle has minimum area.
minimum_rotated_rectangle(geometry, **kwargs)	Computes the oriented envelope (minimum rotated rectangle) that encloses an input geometry, such that the resulting rectangle has minimum area.
minimum_bounding_circle(geometry, **kwargs)	Computes the minimum bounding circle that encloses an input geometry.

5.13.1 shapely.BufferCapStyle

class BufferCapStyle(*value*)

An enumeration.

5.13.2 shapely.BufferJoinStyle

class BufferJoinStyle(*value*)

An enumeration.

5.13.3 shapely.boundary

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

Returns the topological boundary of a geometry.

Parameters

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, LinearRing, LineString, MultiLineString,
↪ MultiPoint, Point, Polygon
>>> boundary(Point(0, 0))
<GEOMETRYCOLLECTION EMPTY>
>>> boundary(LineString([(0, 0), (1, 1), (1, 2)]))
<MULTIPOINT (0 0, 1 2)>
>>> boundary(LinearRing([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]))
<MULTIPOINT EMPTY>
>>> boundary(Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]))
<LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)>
>>> boundary(MultiPoint([(0, 0), (1, 2)]))
<GEOMETRYCOLLECTION EMPTY>
>>> boundary(MultiLineString([[(0, 0), (1, 1)], [(0, 1), (1, 0)]]))
<MULTIPOINT (0 0, 0 1, 1 0, 1 1)>
>>> boundary(GeometryCollection([Point(0, 0)])) is None
True
```

5.13.4 shapely.buffer

buffer(*geometry*, *distance*, *quad_segs=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 distance.

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

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

Parameters

geometry
[Geometry or array_like]

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

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

cap_style
[shapely.BufferCapStyle or {'round', 'square', 'flat'}, default 'round'] Specifies the shape of buffered line endings. BufferCapStyle.round ('round') results in circular line endings (see `quad_segs`). Both BufferCapStyle.square ('square') and BufferCapStyle.flat ('flat') result in rectangular line endings, only BufferCapStyle.flat ('flat') will end at the original vertex, while BufferCapStyle.square ('square') involves adding the buffer width.

join_style
[shapely.BufferJoinStyle or {'round', 'mitre', 'bevel'}, default 'round'] Specifies the shape of buffered line midpoints. BufferJoinStyle.round ('round') results in rounded shapes. BufferJoinStyle.bevel ('bevel') results in a beveled edge that touches the original vertex. BufferJoinStyle.mitre ('mitre') results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, default 5.0] Crops of 'mitre'-style joins if the point is displaced from the buffered vertex by more than this limit.

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point, Polygon, BufferCapStyle, BufferJoinStyle
>>> buffer(Point(10, 10), 2, quad_segs=1)
<POLYGON ((12 10, 10 8, 8 10, 10 12, 12 10))>
>>> buffer(Point(10, 10), 2, quad_segs=2)
<POLYGON ((12 10, 11.414 8.586, 10 8, 8.586 8.586, 8 10, 8.5...>
>>> buffer(Point(10, 10), -2, quad_segs=1)
<POLYGON EMPTY>
```

(continues on next page)

(continued from previous page)

```
>>> line = LineString([(10, 10), (20, 10)])
>>> buffer(line, 2, cap_style="square")
<POLYGON ((20 12, 22 12, 22 8, 10 8, 8 8, 8 12, 20 12))>
>>> buffer(line, 2, cap_style="flat")
<POLYGON ((20 12, 20 8, 10 8, 10 12, 20 12))>
>>> buffer(line, 2, single_sided=True, cap_style="flat")
<POLYGON ((20 10, 10 10, 10 12, 20 12, 20 10))>
>>> line2 = LineString([(10, 10), (20, 10), (20, 20)])
>>> buffer(line2, 2, cap_style="flat", join_style="bevel")
<POLYGON ((18 12, 18 20, 22 20, 22 10, 20 8, 10 8, 10 12, 18 12))>
>>> buffer(line2, 2, cap_style="flat", join_style="mitre")
<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)
<POLYGON ((18 12, 18 20, 22 20, 22 9.172, 20.828 8, 10 8, 10 12, 18 12))>
>>> square = Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)])
>>> buffer(square, 2, join_style="mitre")
<POLYGON ((-2 -2, -2 12, 12 12, 12 -2, -2 -2))>
>>> buffer(square, -2, join_style="mitre")
<POLYGON ((2 2, 2 8, 8 8, 8 2, 2 2))>
>>> buffer(square, -5, join_style="mitre")
<POLYGON EMPTY>
>>> buffer(line, float("nan")) is None
True
```

5.13.5 shapely.offset_curve

offset_curve(*geometry*, *distance*, *quad_segs=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. For a negative distance it will be at the right side. In general, this function tries to preserve the direction of the input.

Note: the behaviour regarding orientation of the resulting line depends on the GEOS version. With GEOS < 3.11, the line retains the same direction for a left offset (positive distance) or has opposite direction for a right offset (negative distance), and this behaviour was documented as such in previous Shapely versions. Starting with GEOS 3.11, the function tries to preserve the orientation of the original line.

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.

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

join_style
[{'round', 'bevel', 'mitre'}, default 'round'] Specifies the shape of outside corners. 'round' results in rounded shapes. 'bevel' results in a beveled edge that touches the original vertex. 'mitre' results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit
[float, default 5.0] Crops of 'mitre'-style joins if the point is displaced from the buffered vertex by more than this limit.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString
>>> line = LineString([(0, 0), (0, 2)])
>>> offset_curve(line, 2)
<LINESTRING (-2 0, -2 2)>
>>> offset_curve(line, -2)
<LINESTRING (2 0, 2 2)>
```

5.13.6 shapely.centroid

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

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

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, MultiPoint, Polygon
>>> centroid(Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]))
<POINT (5 5)>
>>> centroid(LineString([(0, 0), (2, 2), (10, 10)]))
<POINT (5 5)>
>>> centroid(MultiPoint([(0, 0), (10, 10)]))
<POINT (5 5)>
>>> centroid(Polygon())
<POINT EMPTY>
```

5.13.7 shapely.clip_by_rect

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

Returns the portion of a geometry within a rectangle.

The geometry is clipped in a fast but possibly dirty way. The output is not guaranteed to be valid. No exceptions will be raised for topological errors.

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

Parameters

geometry
[Geometry or array_like] The geometry to be clipped

xmin
[float] Minimum x value of the rectangle

ymin
[float] Minimum y value of the rectangle

xmax
[float] Maximum x value of the rectangle

ymax
[float] Maximum y value of the rectangle

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Polygon
>>> line = LineString([(0, 0), (10, 10)])
>>> clip_by_rect(line, 0., 0., 1., 1.)
<LINESTRING (0 0, 1 1)>
>>> polygon = Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)])
>>> clip_by_rect(polygon, 0., 0., 1., 1.)
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
```

5.13.8 shapely.concave_hull

concave_hull(*geometry*, *ratio=0.0*, *allow_holes=False*, ***kwargs*)

Computes a concave geometry that encloses an input geometry.

Note: 'concave_hull' requires at least GEOS 3.11.0.

Parameters

geometry
[Geometry or array_like]

ratio
[float, default 0.0] Number in the range [0, 1]. Higher numbers will include fewer vertices in the hull.

allow_holes
[bool, default False] If set to True, the concave hull may have holes.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import MultiPoint, Polygon
>>> concave_hull(MultiPoint([(0, 0), (0, 3), (1, 1), (3, 0), (3, 3)]), ratio=0.1)
<POLYGON ((0 0, 0 3, 1 1, 3 3, 3 0, 0 0))>
>>> concave_hull(MultiPoint([(0, 0), (0, 3), (1, 1), (3, 0), (3, 3)]), ratio=1.0)
<POLYGON ((0 0, 0 3, 3 3, 3 0, 0 0))>
>>> concave_hull(Polygon())
<POLYGON EMPTY>
```

5.13.9 shapely.convex_hull

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

Computes the minimum convex geometry that encloses an input geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import MultiPoint, Polygon
>>> convex_hull(MultiPoint([(0, 0), (10, 0), (10, 10)]))
<POLYGON ((0 0, 10 10, 10 0, 0 0))>
>>> convex_hull(Polygon())
<GEOMETRYCOLLECTION EMPTY>
```

5.13.10 shapely.delaunay_triangles

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

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

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

Parameters

geometry
[Geometry or array_like]

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, LineString, MultiPoint, Polygon
>>> points = MultiPoint([(50, 30), (60, 30), (100, 100)])
>>> delaunay_triangles(points)
<GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(points, only_edges=True)
<MULTILINESTRING ((50 30, 100 100), (50 30, 60 30), ...>
>>> delaunay_triangles(MultiPoint([(50, 30), (51, 30), (60, 30), (100, 100)]),␣
↪tolerance=2)
<GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Polygon([(50, 30), (60, 30), (100, 100), (50, 30)]))
<GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(LineString([(50, 30), (60, 30), (100, 100)]))
<GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(GeometryCollection([]))
<GEOMETRYCOLLECTION EMPTY>
```

5.13.11 shapely.segmentize

segmentize(*geometry*, *max_segment_length*, ***kwargs*)

Adds vertices to line segments based on maximum segment length.

Note: 'segmentize' requires at least GEOS 3.10.0.

Additional vertices will be added to every line segment in an input geometry so that segments are no longer than the provided maximum segment length. New vertices will evenly subdivide each segment.

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

Parameters

geometry
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Polygon
>>> line = LineString([(0, 0), (0, 10)])
>>> segmentize(line, max_segment_length=5)
<LINESTRING (0 0, 0 5, 0 10)>
>>> polygon = Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)])
>>> segmentize(polygon, max_segment_length=5)
<POLYGON ((0 0, 5 0, 10 0, 10 5, 10 10, 5 10, 0 10, 0 5, 0 0))>
>>> segmentize(None, max_segment_length=5) is None
True
```

5.13.12 shapely.envelope

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

Computes the minimum bounding box that encloses an input geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, LineString, MultiPoint, Point
>>> envelope(LineString([(0, 0), (10, 10)]))
<POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))>
>>> envelope(MultiPoint([(0, 0), (10, 10)]))
<POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))>
>>> envelope(Point(0, 0))
<POINT (0 0)>
>>> envelope(GeometryCollection([]))
<POINT EMPTY>
```

5.13.13 shapely.extract_unique_points

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

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

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, MultiPoint, Point, Polygon
>>> extract_unique_points(Point(0, 0))
<MULTIPOINT (0 0)>
>>> extract_unique_points(LineString([(0, 0), (1, 1), (1, 1)]))
<MULTIPOINT (0 0, 1 1)>
>>> extract_unique_points(Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]))
<MULTIPOINT (0 0, 1 0, 1 1, 0 1)>
>>> extract_unique_points(MultiPoint([(0, 0), (1, 1), (0, 0)]))
<MULTIPOINT (0 0, 1 1)>
>>> extract_unique_points(LineString())
<MULTIPOINT EMPTY>
```

5.13.14 shapely.build_area

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

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

Note: 'build_area' requires at least GEOS 3.8.0.

Equivalent of the PostGIS ST_BuildArea() function.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, Polygon
>>> polygon1 = Polygon([(0, 0), (3, 0), (3, 3), (0, 3), (0, 0)])
>>> polygon2 = Polygon([(1, 1), (1, 2), (2, 2), (1, 1)])
>>> build_area(GeometryCollection([polygon1, polygon2]))
<POLYGON ((0 0, 0 3, 3 3, 3 0, 0 0), (1 1, 2 2, 1 2, 1 1))>
```

5.13.15 shapely.make_valid

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

Repairs invalid geometries.

Note: 'make_valid' requires at least GEOS 3.8.0.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import is_valid, Polygon
>>> polygon = Polygon([(0, 0), (1, 1), (1, 2), (1, 1), (0, 0)])
>>> is_valid(polygon)
False
>>> make_valid(polygon)
<MULTILINESTRING ((0 0, 1 1), (1 1, 1 2))>
```

5.13.16 shapely.normalize

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

Converts Geometry to normal form (or canonical form).

This method orders the coordinates, rings of a polygon and parts of multi geometries consistently. Typically useful for testing purposes (for example in combination with `equals_exact`).

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import MultiLineString
>>> line = MultiLineString([[(0, 0), (1, 1)], [(2, 2), (3, 3)]])
>>> normalize(line)
<MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

5.13.17 shapely.node

node(*geometry*, ***kwargs*)

Returns the fully noded version of the linear input as MultiLineString.

Given a linear input geometry, this function returns a new MultiLineString in which no lines cross each other but only touch at and points. To obtain this, all intersections between segments are computed and added to the segments, and duplicate segments are removed.

Non-linear input (points) will result in an empty MultiLineString.

This function can for example be used to create a fully-noded linework suitable to passed as input to `polygonize`.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point
>>> line = LineString([(0, 0), (1,1), (0, 1), (1, 0)])
>>> node(line)
<MULTILINESTRING ((0 0, 0.5 0.5), (0.5 0.5, 1 1, 0 1, 0.5 0.5), (0.5 0.5, 1 0))>
>>> node(Point(1, 1))
<MULTILINESTRING EMPTY>
```

5.13.18 shapely.point_on_surface

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

Returns a point that intersects an input geometry.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, MultiPoint, Polygon
>>> point_on_surface(Polygon([(0, 0), (10, 0), (10, 10), (0, 10), (0, 0)]))
<POINT (5 5)>
>>> point_on_surface(LineString([(0, 0), (2, 2), (10, 10)]))
<POINT (2 2)>
>>> point_on_surface(MultiPoint([(0, 0), (10, 10)]))
<POINT (0 0)>
>>> point_on_surface(Polygon())
<POINT EMPTY>
```

5.13.19 shapely.polygonize

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 `shapely.multipolygons(shapely.get_parts(shapely.polygonize(geometries)))`.

Parameters

geometries
[array_like] An array of geometries.

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Returns

GeometryCollection or array of GeometryCollections

See also:

get_parts, *get_geometry*
polygonize_full
node

Examples

```
>>> from shapely import LineString
>>> lines = [
...     LineString([(0, 0), (1, 1)]),
...     LineString([(0, 0), (0, 1)]),
...     LineString([(0, 1), (1, 1)])
... ]
>>> polygonize(lines)
<GEOMETRYCOLLECTION (POLYGON ((1 1, 0 0, 0 1, 1 1)))>
```

5.13.20 shapely.polygonize_full

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

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

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

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

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

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

Parameters

geometries
[array_like] An array of geometries.

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Returns

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

See also:

polygonize

Examples

```
>>> from shapely import LineString
>>> lines = [
...     LineString([(0, 0), (1, 1)]),
...     LineString([(0, 0), (0, 1), (1, 1)]),
...     LineString([(0, 1), (1, 1)])
... ]
>>> polygonize_full(lines)
(<GEOMETRYCOLLECTION (POLYGON ((1 1, 0 0, 0 1, 1 1)))>,
 <GEOMETRYCOLLECTION EMPTY>,
 <GEOMETRYCOLLECTION (LINESTRING (0 1, 1 1))>,
 <GEOMETRYCOLLECTION EMPTY>)
```

5.13.21 shapely.remove_repeated_points

remove_repeated_points(*geometry*, *tolerance=0.0*, ***kwargs*)

Returns a copy of a Geometry with repeated points removed.

Note: 'remove_repeated_points' requires at least GEOS 3.11.0.

From the start of the coordinate sequence, each next point within the tolerance is removed.

Removing repeated points with a non-zero tolerance may result in an invalid geometry being returned.

Parameters

geometry
[Geometry or array_like]

tolerance
[float or array_like, default=0.0] Use 0.0 to remove only exactly repeated points.

Examples

```
>>> from shapely import LineString, Polygon
>>> remove_repeated_points(LineString([(0,0), (0,0), (1,0)]), tolerance=0)
<LINESTRING (0 0, 1 0)>
>>> remove_repeated_points(Polygon([(0, 0), (0, .5), (0, 1), (.5, 1), (0,0)]),␣
↪tolerance=.5)
<POLYGON ((0 0, 0 1, 0 0, 0 0))>
```

5.13.22 shapely.reverse

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

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

Note: 'reverse' requires at least GEOS 3.7.0.

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

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

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

is_ccw
Checks if a Geometry is clockwise.

Examples

```
>>> from shapely import LineString, Polygon
>>> reverse(LineString([(0, 0), (1, 2)]))
<LINESTRING (1 2, 0 0)>
>>> reverse(Polygon([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]))
<POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>
>>> reverse(None) is None
True
```

5.13.23 shapely.simplify

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

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

Parameters

geometry
[Geometry or array_like]

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

preserve_topology
[bool, default True] By default (True), the operation will avoid creating invalid geometries (checking for collapses, ring-intersections, etc), but this is computationally more expensive.

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Polygon
>>> line = LineString([(0, 0), (1, 10), (0, 20)])
>>> simplify(line, tolerance=0.9)
<LINESTRING (0 0, 1 10, 0 20)>
>>> simplify(line, tolerance=1)
<LINESTRING (0 0, 0 20)>
>>> polygon_with_hole = Polygon(
...     [(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)],
...     holes=[[(2, 2), (2, 4), (4, 4), (4, 2), (2, 2)]]
... )
>>> simplify(polygon_with_hole, tolerance=4, preserve_topology=True)
<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)
<POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
```

5.13.24 shapely.snap

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

Snaps an input geometry to reference geometry's vertices.

Vertices of the first geometry are snapped to vertices of the second. geometry, returning a new geometry; the input geometries are not modified. The result geometry is the input geometry with the vertices snapped. If no snapping occurs then the input geometry is returned unchanged. The tolerance is used to control where snapping is performed.

Where possible, this operation tries to avoid creating invalid geometries; however, it does not guarantee that output geometries will be valid. It is the responsibility of the caller to check for and handle invalid geometries.

Because too much snapping can result in invalid geometries being created, heuristics are used to determine the number and location of snapped vertices that are likely safe to snap. These heuristics may omit some potential snaps that are otherwise within the tolerance.

Parameters

geometry
[Geometry or array_like]

reference
[Geometry or array_like]

tolerance
[float or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import snap, distance, LineString, Point, Polygon, MultiPoint, box
```

```
>>> point = Point(0.5, 2.5)
>>> target_point = Point(0, 2)
>>> snap(point, target_point, tolerance=1)
<POINT (0 2)>
>>> snap(point, target_point, tolerance=0.49)
<POINT (0.5 2.5)>
```

```
>>> polygon = Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)])
>>> snap(polygon, Point(8, 10), tolerance=5)
<POLYGON ((0 0, 0 10, 8 10, 10 0, 0 0))>
>>> snap(polygon, LineString([(8, 10), (8, 0)]), tolerance=5)
<POLYGON ((0 0, 0 10, 8 10, 8 0, 0 0))>
```

You can snap one line to another, for example to clean imprecise coordinates:

```
>>> line1 = LineString([(0.1, 0.1), (0.49, 0.51), (1.01, 0.89)])
>>> line2 = LineString([(0, 0), (0.5, 0.5), (1.0, 1.0)])
>>> snap(line1, line2, 0.25)
<LINESTRING (0 0, 0.5 0.5, 1 1)>
```

Snapping also supports Z coordinates:

```
>>> point1 = Point(0.1, 0.1, 0.5)
>>> multipoint = MultiPoint([(0, 0, 1), (0, 0, 0)])
>>> snap(point1, multipoint, 1)
<POINT Z (0 0 1)>
```

Snapping to an empty geometry has no effect:

```
>>> snap(line1, LineString([]), 0.25)
<LINESTRING (0.1 0.1, 0.49 0.51, 1.01 0.89)>
```

Snapping to a non-geometry (None) will always return None:

```
>>> snap(line1, None, 0.25) is None
True
```

Only one vertex of a polygon is snapped to a target point, even if all vertices are equidistant to it, in order to prevent collapse of the polygon:

```
>>> poly = box(0, 0, 1, 1)
>>> poly
<POLYGON ((1 0, 1 1, 0 1, 0 0, 1 0))>
>>> snap(poly, Point(0.5, 0.5), 1)
<POLYGON ((0.5 0.5, 1 1, 0 1, 0 0, 0.5 0.5))>
```

5.13.25 shapely.voronoi_polygons

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

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

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

Parameters

geometry
[Geometry or array_like]

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, MultiPoint, normalize, Point
>>> points = MultiPoint([(2, 2), (4, 2)])
>>> normalize(voronoi_polygons(points))
<GEOMETRYCOLLECTION (POLYGON ((3 0, 3 4, 6 4, 6 0, 3 0)), POLYGON ((0 0, 0 4...>
>>> voronoi_polygons(points, only_edges=True)
<LINESTRING (3 4, 3 0)>
>>> voronoi_polygons(MultiPoint([(2, 2), (4, 2), (4.2, 2)]), 0.5, only_edges=True)
<LINESTRING (3 4.2, 3 -0.2)>
>>> voronoi_polygons(points, extend_to=LineString([(0, 0), (10, 10)]), only_
↪edges=True)
<LINESTRING (3 10, 3 0)>
>>> voronoi_polygons(LineString([(2, 2), (4, 2)]), only_edges=True)
<LINESTRING (3 4, 3 0)>
>>> voronoi_polygons(Point(2, 2))
<GEOMETRYCOLLECTION EMPTY>
```

5.13.26 shapely.oriented_envelope

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

Computes the oriented envelope (minimum rotated rectangle) that encloses an input geometry, such that the resulting rectangle has minimum area.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, LineString, MultiPoint, Point, Polygon
>>> oriented_envelope(MultiPoint([(0, 0), (10, 0), (10, 10)])).normalize()
<POLYGON ((0 0, 10 10, 15 5, 5 -5, 0 0))>
>>> oriented_envelope(LineString([(1, 1), (5, 1), (10, 10)])).normalize()
<POLYGON ((1 1, 10 10, 12 8, 3 -1, 1 1))>
>>> oriented_envelope(Polygon([(1, 1), (15, 1), (5, 10), (1, 1)])).normalize()
<POLYGON ((1 1, 1 10, 15 10, 15 1, 1 1))>
>>> oriented_envelope(LineString([(1, 1), (10, 1)])).normalize()
<LINESTRING (1 1, 10 1)>
>>> oriented_envelope(Point(2, 2))
<POINT (2 2)>
>>> oriented_envelope(GeometryCollection([]))
<POLYGON EMPTY>
```

5.13.27 shapely.minimum_rotated_rectangle

minimum_rotated_rectangle(*geometry*, ***kwargs*)

Computes the oriented envelope (minimum rotated rectangle) that encloses an input geometry, such that the resulting rectangle has minimum area.

Unlike envelope this rectangle is not constrained to be parallel to the coordinate axes. If the convex hull of the object is a degenerate (line or point) this degenerate is returned.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import GeometryCollection, LineString, MultiPoint, Point, Polygon
>>> oriented_envelope(MultiPoint([(0, 0), (10, 0), (10, 10)])).normalize()
<POLYGON ((0 0, 10 10, 15 5, 5 -5, 0 0))>
>>> oriented_envelope(LineString([(1, 1), (5, 1), (10, 10)])).normalize()
<POLYGON ((1 1, 10 10, 12 8, 3 -1, 1 1))>
>>> oriented_envelope(Polygon([(1, 1), (15, 1), (5, 10), (1, 1)])).normalize()
<POLYGON ((1 1, 1 10, 15 10, 15 1, 1 1))>
>>> oriented_envelope(LineString([(1, 1), (10, 1)])).normalize()
<LINESTRING (1 1, 10 1)>
>>> oriented_envelope(Point(2, 2))
<POINT (2 2)>
>>> oriented_envelope(GeometryCollection([]))
<POLYGON EMPTY>
```

5.13.28 shapely.minimum_bounding_circle

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

Computes the minimum bounding circle that encloses an input geometry.

Note: 'minimum_bounding_circle' requires at least GEOS 3.8.0.

Parameters

geometry
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

minimum_bounding_radius

Examples

```
>>> from shapely import GeometryCollection, LineString, MultiPoint, Point, Polygon
>>> minimum_bounding_circle(Polygon([(0, 0), (0, 10), (10, 10), (10, 0), (0, 0)]))
<POLYGON ((12.071 5, 11.935 3.621, 11.533 2.294, 10.879 1.07...>
>>> minimum_bounding_circle(LineString([(1, 1), (10, 10)]))
<POLYGON ((11.864 5.5, 11.742 4.258, 11.38 3.065, 10.791 1.9...>
>>> minimum_bounding_circle(MultiPoint([(2, 2), (4, 2)]))
<POLYGON ((4 2, 3.981 1.805, 3.924 1.617, 3.831 1.444, 3.707...>
>>> minimum_bounding_circle(Point(0, 1))
<POINT (0 1)>
>>> minimum_bounding_circle(GeometryCollection([]))
<POLYGON EMPTY>
```

5.14 Linestring operations

line_interpolate_point(line, distance[, ...])	Returns a point interpolated at given distance on a line.
line_locate_point(line, other[, normalized])	Returns the distance to the line origin of given point.
line_merge(line[, directed])	Returns (Multi)LineStrings formed by combining the lines in a MultiLineString.
shared_paths(a, b, **kwargs)	Returns the shared paths between geom1 and geom2.
shortest_line(a, b, **kwargs)	Returns the shortest line between two geometries.

5.14.1 shapely.line_interpolate_point

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

Returns a point interpolated at given distance on a line.

Parameters

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

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

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point
>>> line = LineString([(0, 2), (0, 10)])
>>> line_interpolate_point(line, 2)
<POINT (0 4)>
>>> line_interpolate_point(line, 100)
<POINT (0 10)>
>>> line_interpolate_point(line, -2)
<POINT (0 8)>
>>> line_interpolate_point(line, [0.25, -0.25], normalized=True).tolist()
[<POINT (0 4)>, <POINT (0 8)>]
>>> line_interpolate_point(LineString(), 1)
<POINT EMPTY>
```

5.14.2 shapely.line_locate_point

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

Returns the distance to the line origin of given point.

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

Parameters

line
[Geometry or array_like]

point
[Geometry or array_like]

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

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString, Point
>>> line = LineString([(0, 2), (0, 10)])
>>> point = Point(4, 4)
>>> line_locate_point(line, point)
2.0
>>> line_locate_point(line, point, normalized=True)
0.25
>>> line_locate_point(line, Point(0, 18))
8.0
>>> line_locate_point(LineString(), point)
nan
```

5.14.3 shapely.line_merge

line_merge(*line*, *directed=False*, ***kwargs*)

Returns (Multi)LineStrings formed by combining the lines in a MultiLineString.

Lines are joined together at their endpoints in case two lines are intersecting. Lines are not joined when 3 or more lines are intersecting at the endpoints. Line elements that cannot be joined are kept as is in the resulting MultiLineString.

The direction of each merged LineString will be that of the majority of the LineStrings from which it was derived. Except if `directed=True` is specified, then the operation will not change the order of points within lines and so only lines which can be joined with no change in direction are merged.

Parameters

line
[Geometry or array_like]

directed
[bool, default False] Only combine lines if possible without changing point order. Requires GEOS >= 3.11.0

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import MultiLineString
>>> line_merge(MultiLineString([[(0, 2), (0, 10)], [(0, 10), (5, 10)]]))
<LINESTRING (0 2, 0 10, 5 10)>
>>> line_merge(MultiLineString([[(0, 2), (0, 10)], [(0, 11), (5, 10)]]))
<MULTILINESTRING ((0 2, 0 10), (0 11, 5 10))>
>>> line_merge(MultiLineString())
<GEOMETRYCOLLECTION EMPTY>
>>> line_merge(MultiLineString([[(0, 0), (1, 0)], [(0, 0), (3, 0)]]))
<LINESTRING (1 0, 0 0, 3 0)>
>>> line_merge(MultiLineString([[(0, 0), (1, 0)], [(0, 0), (3, 0)]]), directed=True)
<MULTILINESTRING ((0 0, 1 0), (0 0, 3 0))>
```

5.14.4 shapely.shared_paths

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

Returns the shared paths between geom1 and geom2.

Both geometries should be linestrings or arrays of linestrings. A geometrycollection or array of geometrycollections is returned with two elements in each geometrycollection. The first element is a multilinestring containing shared paths with the same direction for both inputs. The second element is a multilinestring containing shared paths with the opposite direction for the two inputs.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

Examples

```
>>> from shapely import LineString
>>> line1 = LineString([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> line2 = LineString([(1, 0), (2, 0), (2, 1), (1, 1), (1, 0)])
>>> shared_paths(line1, line2).wkt
'GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MULTILINESTRING ((1 0, 1 1)))'
>>> line3 = LineString([(1, 1), (0, 1)])
>>> shared_paths(line1, line3).wkt
'GEOMETRYCOLLECTION (MULTILINESTRING ((1 1, 0 1)), MULTILINESTRING EMPTY)'
```

5.14.5 shapely.shortest_line

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

Returns the shortest line between two geometries.

The resulting line consists of two points, representing the nearest points between the geometry pair. The line always starts in the first geometry *a* and ends in he second geometry *b*. The endpoints of the line will not necessarily be existing vertices of the input geometries *a* and *b*, but can also be a point along a line segment.

Parameters

a
[Geometry or array_like]

b
[Geometry or array_like]

****kwargs**
See NumPy ufunc docs for other keyword arguments.

See also:

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

Examples

```
>>> from shapely import LineString
>>> line1 = LineString([(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)])
>>> line2 = LineString([(0, 3), (3, 0), (5, 3)])
>>> shortest_line(line1, line2)
<LINESTRING (1 1, 1.5 1.5)>
```

5.15 Coordinate operations

transform(geometry, transformation[, include_z])	Returns a copy of a geometry array with a function applied to its coordinates.
count_coordinates(geometry)	Counts the number of coordinate pairs in a geometry array.
get_coordinates(geometry[, include_z, ...])	Gets coordinates from a geometry array as an array of floats.
set_coordinates(geometry, coordinates)	Adapts the coordinates of a geometry array in-place.

5.15.1 shapely.transform

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

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

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

Parameters

geometry
[Geometry or array_like]

transformation
[function] A function that transforms a (N, 2) or (N, 3) ndarray of float64 to another (N, 2) or (N, 3) ndarray of float64.

include_z
[bool, default False] If True, include the third dimension in the coordinates array that is passed to the `transformation` function. If a geometry has no third dimension, the z-coordinates passed to the function will be NaN.

Examples

```
>>> from shapely import LineString, Point
>>> transform(Point(0, 0), lambda x: x + 1)
<POINT (1 1)>
>>> transform(LineString([(2, 2), (4, 4)]), lambda x: x * [2, 3])
<LINESTRING (4 6, 8 12)>
>>> transform(None, lambda x: x) is None
True
>>> transform([Point(0, 0), None], lambda x: x).tolist()
[<POINT (0 0)>, None]
```

By default, the third dimension is ignored:

```
>>> transform(Point(0, 0, 0), lambda x: x + 1)
<POINT (1 1)>
>>> transform(Point(0, 0, 0), lambda x: x + 1, include_z=True)
<POINT Z (1 1 1)>
```

5.15.2 shapely.count_coordinates

count_coordinates(*geometry*)

Counts the number of coordinate pairs in a geometry array.

Parameters

geometry
[Geometry or array_like]

Examples

```
>>> from shapely import LineString, Point
>>> count_coordinates(Point(0, 0))
1
>>> count_coordinates(LineString([(2, 2), (4, 2)]))
2
>>> count_coordinates(None)
0
>>> count_coordinates([Point(0, 0), None])
1
```

5.15.3 shapely.get_coordinates

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

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

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

Parameters

geometry
[Geometry or array_like]

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

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

Examples

```
>>> from shapely import LineString, Point
>>> get_coordinates(Point(0, 0)).tolist()
[[0.0, 0.0]]
>>> get_coordinates(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(Point(0, 0, 0)).tolist()
[[0.0, 0.0]]
>>> get_coordinates(Point(0, 0, 0), include_z=True).tolist()
[[0.0, 0.0, 0.0]]
```

When return_index=True, indexes are returned also:

```
>>> geometries = [LineString([(2, 2), (4, 4)]), Point(0, 0)]
>>> coordinates, index = get_coordinates(geometries, return_index=True)
>>> coordinates.tolist(), index.tolist()
([[2.0, 2.0], [4.0, 4.0], [0.0, 0.0]], [0, 0, 1])
```

5.15.4 shapely.set_coordinates

set_coordinates(*geometry*, *coordinates*)

Adapts the coordinates of a geometry array in-place.

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

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

Parameters

geometry
[Geometry or array_like]

coordinates: array_like

See also:

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

Examples

```
>>> from shapely import LineString, Point
>>> set_coordinates(Point(0, 0), [[1, 1]])
<POINT (1 1)>
>>> set_coordinates([Point(0, 0), LineString([(0, 0), (0, 0)])], [[1, 2], [3, 4],␣
↪[5, 6]]).tolist()
[<POINT (1 2)>, <LINESTRING (3 4, 5 6)>]
>>> set_coordinates([None, Point(0, 0)], [[1, 2]]).tolist()
[None, <POINT (1 2)>]
```

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

```
>>> set_coordinates(Point(0, 0, 0), [[1, 1]])
<POINT (1 1)>
>>> set_coordinates(Point(0, 0, 0), [[1, 1, 1]])
<POINT Z (1 1 1)>
```

5.16 STRTree

class STRtree(*geoms: Iterable[BaseGeometry]*, *node_capacity: int = 10*)

A query-only R-tree spatial index created using the Sort-Tile-Recursive (STR) [1] algorithm.

The tree indexes the bounding boxes of each geometry. The tree is constructed directly at initialization and nodes cannot be added or removed after it has been created.

All operations return indices of the input geometries. These indices can be used to index into anything associated with the input geometries, including the input geometries themselves, or custom items stored in another object of the same length as the geometries.

Bounding boxes limited to two dimensions and are axis-aligned (equivalent to the `bounds` property of a geometry); any Z values present in geometries are ignored for purposes of indexing within the tree.

Any mixture of geometry types may be stored in the tree.

Note: the tree is more efficient for querying when there are fewer geometries that have overlapping bounding boxes and where there is greater similarity between the outer boundary of a geometry and its bounding box. For example, a MultiPolygon composed of widely-spaced individual Polygons will have a large overall bounding box compared to the boundaries of its individual Polygons, and the bounding box may also potentially overlap many other geometries within the tree. This means that the resulting tree may be less efficient to query than a tree constructed from individual Polygons.

Parameters

geoms
[sequence] A sequence of geometry objects.

node_capacity
[int, default 10] The maximum number of child nodes per parent node in the tree.

References

[1]

property geometries

Geometries stored in the tree in the order used to construct the tree.

The order of this array corresponds to the tree indices returned by other STRtree methods.

Do not attempt to modify items in the returned array.

Returns

ndarray of Geometry objects

nearest(*geometry*) → Any | None

Return the index of the nearest geometry in the tree for each input geometry based on distance within two-dimensional Cartesian space.

Note: 'nearest' requires at least GEOS 3.6.0.

This distance will be 0 when input geometries intersect tree geometries.

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.

If any input geometry is None or empty, an error is raised. Any Z values present in input geometries are ignored when finding nearest tree geometries.

Parameters

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

Returns

scalar or ndarray
Indices of geometries in tree. Return value will have the same shape as the input.

None is returned if this index is empty. This may change in version 2.0.

See also:

query_nearest
returns all equidistant geometries, exclusive geometries, and optional distances

Examples

```
>>> from shapely.geometry import Point
>>> tree = STRtree([Point(i, i) for i in range(10)])
```

Query the tree for nearest using a scalar geometry:

```
>>> index = tree.nearest(Point(2.2, 2.2))
>>> index
2
>>> tree.geometries.take(index)
<POINT (2 2)>
```

Query the tree for nearest using an array of geometries:

```
>>> indices = tree.nearest([Point(2.2, 2.2), Point(4.4, 4.4)])
>>> indices.tolist()
[2, 4]
>>> tree.geometries.take(indices).tolist()
[<POINT (2 2)>, <POINT (4 4)>]
```

Nearest only return one object if there are multiple equidistant results:

```
>>> tree = STRtree ([Point(0, 0), Point(0, 0)])
>>> tree.nearest(Point(0, 0))
0
```

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

Return the integer indices of all combinations of each input geometry and tree geometries where the bounding box of each input geometry intersects the bounding box of a tree geometry.

If the input geometry is a scalar, this returns an array of shape (n,) with the indices of the matching tree geometries. If the input geometry is an array_like, this returns an array with shape (2,n) where the subarrays correspond to the indices of the input geometries and indices of the tree geometries associated with each. To generate an array of pairs of input geometry index and tree geometry index, simply transpose the result.

If a predicate is provided, the tree geometries are first queried based on the bounding box of the input geometry and then are further filtered to those that meet the predicate when comparing the input geometry to the tree geometry: predicate(geometry, tree_geometry)

The 'dwithin' predicate requires GEOS >= 3.10.

Bounding boxes are limited to two dimensions and are axis-aligned (equivalent to the `bounds` property of a geometry); any Z values present in input geometries are ignored when querying the tree.

Any input geometry that is None or empty will never match geometries in the tree.

Parameters

geometry
[Geometry or array_like] Input geometries to query the tree and filter results using the optional predicate.

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

distance
[number or array_like, optional] Distances around each input geometry within which to query the tree for the 'dwithin' predicate. If array_like, shape must be broadcastable to shape of geometry. Required if predicate='dwithin'.

Returns

ndarray with shape (n,) if geometry is a scalar
Contains tree geometry indices.

OR
ndarray with shape (2, n) if geometry is an array_like
The first subarray contains input geometry indices. The second subarray contains tree geometry indices.

Notes

In the context of a spatial join, input geometries are the "left" geometries that determine the order of the results, and tree geometries are "right" geometries that are joined against the left geometries. This effectively performs an inner join, where only those combinations of geometries that can be joined based on overlapping bounding boxes or optional predicate are returned.

Examples

```
>>> from shapely import box, Point
>>> import numpy as np
>>> points = [Point(0, 0), Point(1, 1), Point(2,2), Point(3, 3)]
>>> tree = STRtree(points)
```

Query the tree using a scalar geometry:

```
>>> indices = tree.query(box(0, 0, 1, 1))
>>> indices.tolist()
[0, 1]
```

Query using an array of geometries:

```
>>> boxes = np.array([box(0, 0, 1, 1), box(2, 2, 3, 3)])
>>> arr_indices = tree.query(boxes)
>>> arr_indices.tolist()
[[0, 0, 1, 1], [0, 1, 2, 3]]
```

Or transpose to get all pairs of input and tree indices:

```
>>> arr_indices.T.tolist()
[[0, 0], [0, 1], [1, 2], [1, 3]]
```

Retrieve the tree geometries by results of query:

```
>>> tree.geometries.take(indices).tolist()
[<POINT (0 0)>, <POINT (1 1)>]
```

Retrieve all pairs of input and tree geometries:

```
>>> np.array([boxes.take(arr_indices[0]),tree.geometries.take(arr_indices[1])]).
↪T.tolist()
[[<POLYGON ((1 0, 1 1, 0 1, 0 0, 1 0))>, <POINT (0 0)>],
 [<POLYGON ((1 0, 1 1, 0 1, 0 0, 1 0))>, <POINT (1 1)>],
 [<POLYGON ((3 2, 3 3, 2 3, 2 2, 3 2))>, <POINT (2 2)>],
 [<POLYGON ((3 2, 3 3, 2 3, 2 2, 3 2))>, <POINT (3 3)>]]
```

Query using a predicate:

```
>>> tree = STRtree([box(0, 0, 0.5, 0.5), box(0.5, 0.5, 1, 1), box(1, 1, 2, 2)])
>>> tree.query(box(0, 0, 1, 1), predicate="contains").tolist()
[0, 1]
>>> tree.query(Point(0.75, 0.75), predicate="dwithin", distance=0.5).tolist()
[0, 1, 2]
```

```
>>> tree.query(boxes, predicate="contains").tolist()
[[0, 0], [0, 1]]
>>> tree.query(boxes, predicate="dwithin", distance=0.5).tolist()
[[0, 0, 0, 1], [0, 1, 2, 2]]
```

Retrieve custom items associated with tree geometries (records can be in whatever data structure so long as geometries and custom data can be extracted into arrays of the same length and order):

```
>>> records = [
...     {"geometry": Point(0, 0), "value": "A"},
...     {"geometry": Point(2, 2), "value": "B"}
... ]
>>> tree = STRtree([record["geometry"] for record in records])
>>> items = np.array([record["value"] for record in records])
>>> items.take(tree.query(box(0, 0, 1, 1))).tolist()
['A']
```

query_nearest(*geometry*, *max_distance=None*, *return_distance=False*, *exclusive=False*, *all_matches=True*)

Return the index of the nearest geometries in the tree for each input geometry based on distance within two-dimensional Cartesian space.

Note: 'query_nearest' requires at least GEOS 3.6.0.

This distance will be 0 when input geometries intersect tree geometries.

If there are multiple equidistant or intersected geometries in tree and *all_matches* is True (the default), all matching tree geometries are returned; otherwise only the first matching tree geometry is returned. Tree indices 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 geometry within +/- max_distance beyond their envelope will be evaluated. However, using a large max_distance may have a negative performance impact because many tree geometries will be queried for each input geometry.

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. Any Z values present in input geometries are ignored when finding nearest tree geometries.

Parameters

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

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

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

exclusive
[bool, default False] If True, the nearest tree geometries that are equal to the input geometry will not be returned.

all_matches
[bool, default True] If True, all equidistant and intersected geometries will be returned for each input geometry. If False, only the first nearest geometry will be returned.

Returns

tree indices or tuple of (tree indices, distances) if geometry is a scalar
indices is an ndarray of shape (n,) and distances (if present) an ndarray of shape (n,)

OR
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 numpy as np
>>> from shapely import box, Point
>>> points = [Point(0, 0), Point(1, 1), Point(2,2), Point(3, 3)]
>>> tree = STRtree(points)
```

Find the nearest tree geometries to a scalar geometry:

```
>>> indices = tree.query_nearest(Point(0.25, 0.25))
>>> indices.tolist()
[0]
```

Retrieve the tree geometries by results of query:

```
>>> tree.geometries.take(indices).tolist()
[<POINT (0 0)>]
```

Find the nearest tree geometries to an array of geometries:

```
>>> query_points = np.array([Point(2.25, 2.25), Point(1, 1)])
>>> arr_indices = tree.query_nearest(query_points)
>>> arr_indices.tolist()
[[0, 1], [2, 1]]
```

Or transpose to get all pairs of input and tree indices:

```
>>> arr_indices.T.tolist()
[[0, 2], [1, 1]]
```

Retrieve all pairs of input and tree geometries:

```
>>> list(zip(query_points.take(arr_indices[0]), tree.geometries.take(arr_
↪indices[1])))
[(<POINT (2.25 2.25)>, <POINT (2 2)>), (<POINT (1 1)>, <POINT (1 1)>)]
```

All intersecting geometries in the tree are returned by default:

```
>>> tree.query_nearest(box(1,1,3,3)).tolist()
[1, 2, 3]
```

Set all_matches to False to to return a single match per input geometry:

```
>>> tree.query_nearest(box(1,1,3,3), all_matches=False).tolist()
[1]
```

Return the distance to each nearest tree geometry:

```
>>> index, distance = tree.query_nearest(Point(0.5, 0.5), return_distance=True)
>>> index.tolist()
[0, 1]
>>> distance.round(4).tolist()
[0.7071, 0.7071]
```

Return the distance for each input and nearest tree geometry for an array of geometries:

```
>>> indices, distance = tree.query_nearest([Point(0.5, 0.5), Point(1, 1)],␣
↪return_distance=True)
>>> indices.tolist()
[[0, 0, 1], [0, 1, 1]]
>>> distance.round(4).tolist()
[0.7071, 0.7071, 0.0]
```

Retrieve custom items associated with tree geometries (records can be in whatever data structure so long as geometries and custom data can be extracted into arrays of the same length and order):

```
>>> records = [
...     {"geometry": Point(0, 0), "value": "A"},
...     {"geometry": Point(2, 2), "value": "B"}
... ]
>>> tree = STRtree([record["geometry"] for record in records])
>>> items = np.array([record["value"] for record in records])
>>> items.take(tree.query_nearest(Point(0.5, 0.5))).tolist()
['A']
```

5.17 Testing

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

```
>>> from shapely.testing import assert_geometries_equal
```

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

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

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

Parameters

x
[Geometry or array_like]

y
[Geometry or array_like]

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

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

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

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

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

5.18 Indices and tables

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Leutenegger, Scott T.; Edgington, Jeffrey M.; Lopez, Mario A. (February 1997). “STR: A Simple and Efficient Algorithm for R-Tree Packing”. https://ia600900.us.archive.org/27/items/nasa_techdoc_19970016975/19970016975.pdf

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S