Shapely Documentation

Release 1.6.4.post1

Sean Gillies

Mar 21, 2018

Contents

1 Documentation Contents **1**
1.1 Shapely 1
1.2 The Shapely User Manual 17

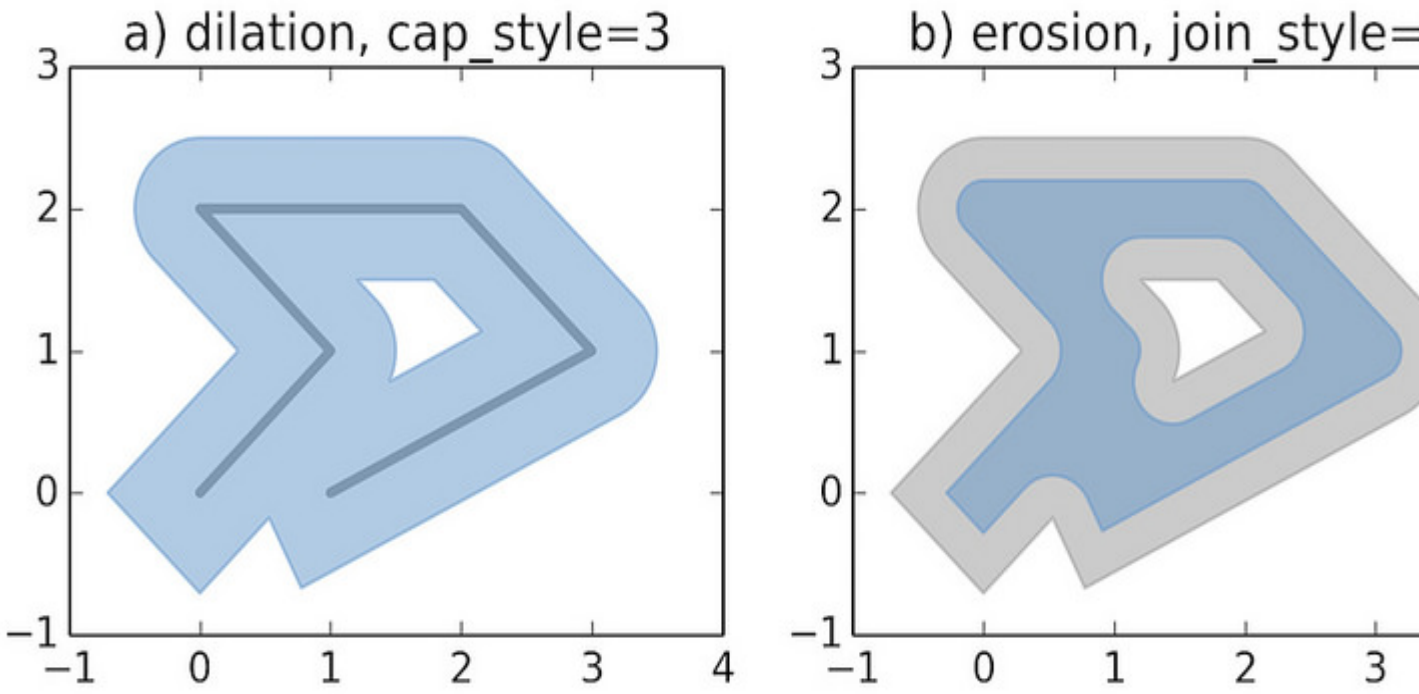
2 Indices and tables **65**

CHAPTER 1

Documentation Contents

1.1 Shapely

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 based on the widely deployed GEOS (the engine of PostGIS) and JTS (from which GEOS is ported) libraries. Shapely is not concerned with data formats or coordinate systems, but can be readily integrated with packages that are. For more

details, see:

- Shapely GitHub repository
- Shapely documentation and manual

1.1.1 Requirements

Shapely 1.6 requires

- Python 2.7, >=3.4
- GEOS >=3.3

1.1.2 Installing Shapely 1.6

Shapely may be installed from a source distribution or one of several kinds of built distribution.

Built distributions

Windows users have two good installation options: the wheels at http://www.lfd.uci.edu/~gohlke/pythonlibs/#shapely and the Anaconda platform's conda-forge channel.

OS X and Linux users can get Shapely wheels with GEOS included from the Python Package Index with a recent version of pip (8+):

```
$ pip install shapely
```

A few extra speedups that require Numpy can be had by running

```
$ pip install shapely[vectorized]
```

Shapely is available via system package management tools like apt, yum, and Homebrew, and is also provided by popular Python distributions like Canopy and Anaconda.

Source distributions

If you want to build Shapely from source for compatibility with other modules that depend on GEOS (such as cartopy or osgeo.ogr) or want to use a different version of GEOS than the one included in the project wheels you should first install the GEOS library, Cython, and Numpy on your system (using apt, yum, brew, or other means) and then direct pip to ignore the binary wheels.

```
$ pip install shapely --no-binary shapely
```

If you've installed GEOS to a standard location, the geos-config program will be used to get compiler and linker options. If geos-config is not on your executable, it can be specified with a GEOS_CONFIG environment variable, e.g.:

```
$ GEOS_CONFIG=/path/to/geos-config pip install shapely
```

1.1.3 Usage

Here is the canonical example of building an approximately circular patch by buffering a point.

```
>>> from shapely.geometry import Point
>>> patch = Point(0.0, 0.0).buffer(10.0)
>>> patch
<shapely.geometry.polygon.Polygon object at 0x...>
>>> patch.area
313.65484905459385
```

See the manual for comprehensive usage snippets and the dissolve.py and intersect.py examples.

1.1.4 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
<shapely.geometry.point.Point object at 0x...>
>>> print(json.dumps(mapping(s)))
{"type": "Point", "coordinates": [0.0, 0.0]}
```

1.1.5 Development and Testing

Dependencies for developing Shapely are listed in requirements-dev.txt. Cython and Numpy are not required for production installations, only for development. Use of a virtual environment is strongly recommended.

```
$ virtualenv .
$ source bin/activate
(env)$ pip install -r requirements-dev.txt
(env)$ pip install -e .
```

We use py.test to run Shapely's suite of unittests and doctests.

```
(env)$ python -m pytest
```

1.1.6 Support

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

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

1.1.7 Credits

Shapely is written by:

- Sean Gillies <sean.gillies@gmail.com>
- Oliver Tonnhofer <olt@bogosoft.com>
- Joshua Arnott <josh@snorfalorpagus.net>
- Mike Toews <mwtoews@gmail.com>
- Jacob Wasserman <jwasserman@gmail.com>
- Aron Bierbaum <aronbierbaum@gmail.com>
- Allan Adair <allan@rfspot.com>
- Johannes Schönberger <jschoenberger@demuc.de>
- georgeouzou <geothrock@gmail.com>
- Phil Elson <pelson.pub@gmail.com>
- Howard Butler <hobu.inc@gmail.com>
- Kelsey Jordahl <kjordahl@enthought.com>
- dokai <dokai@b426a367-1105-0410-b9ff-cdf4ab011145>
- Kevin Wurster <kevin@skytruth.org>
- Gabi Davar <grizzly.nyo@gmail.com>
- Thibault Deutsch <thibault.deutsch@gmail.com>
- Dave Collins <dave@hopest.net>
- fredj <frederic.junod@camptocamp.com>
- Brad Hards <bradh@frogmouth.net>
- David Baumgold <david@davidbaumgold.com>
- Henry Walshaw <henry.walshaw@gmail.com>
- Jinkun Wang <mejkunw@gmail.com>
- Marc Jansen <jansen@terrestris.de>
- Sampo Syrjanen <sampo.syrjanen@here.com>
- Steve M. Kim <steve@climate.com>
- Thomas Kluyver <takowl@gmail.com>
- Morris Tweed <tweed.morris@gmail.com>
- Naveen Michaud-Agrawal <naveen.michaudagrawal@gmail.com>
- Jeethu Rao <jeethu@jeethurao.com>
- Peter Sagerson <psagers.github@ignorare.net>
- Jason Sanford <jason.sanford@mapmyfitness.com>
- mindw <grizzly.nyo@gmail.com>
- Jamie Hall <jamie1212@gmail.com>
- James Spencer <james.s.spencer@gmail.com>

- Stephan Hügel <urschrei@gmail.com>
- Bas Couwenberg <sebastic@xs4all.nl>
- James Douglass <jamesdouglassusa@gmail.com>
- Tobias Sauerwein <tobias.sauerwein@camptocamp.com>
- WANG Aiyong <gepcelway@gmail.com>
- Brandon Wood <btwood@geometeor.com>
- BertrandGervais <bertrand.gervais.pro@gmail.com>
- Andy Freeland <andy@andyfreeland.net>
- Benjamin Root <ben.v.root@gmail.com>
- giumas <gmasetti@ccom.unh.edu>
- Leandro Lima <leandro@limaesilva.com.br>
- Maarten Vermeyen <maarten.vermeyen@rwo.vlaanderen.be>
- joelostblom <joelostblom@users.noreply.github.com>
- Marco De Nadai <me@marcodena.it>
- Johan Euphrosine <proppy@aminche.com>

See also: https://github.com/Toblerity/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 (http://www.neh.gov).

1.1.8 Changes

Next

New features:

- Use DLLs indicated in sys._MEIPASS' to support PyInstaller frozen apps (#523).

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

1.2 The Shapely User Manual

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

Version 1.6.4

Date Mar 21, 2018

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

1.2.1 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 (via Python's `ctypes` module) 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.

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

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

[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.

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 http://www.vividsolutions.com/jts/discussion.htm#spatialDataModel 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[2]: 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.

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.

1.2.2 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.

[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]

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

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 suprisingly invalid geometry objects. For example, `LineString([(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, 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.

`object.`**`geom_type`**
Returns a string specifying the *Geometry Type* of the object in accordance with[1].

```
>>> print 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 is the furthest distance from any point on the first geometry to any point on the second 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.wkt
'POINT (-0.0000000000000001 -0.0000000000000000)'
>>> donut.representative_point().wkt
'POINT (-1.5000000000000000 0.0000000000000000)'
```

Points

class Point(*coordinates*)

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

```
>>> from shapely.geometry 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)
<shapely.geometry.point.Point object at 0x...>
```

LineStrings

class LineString(*coordinates*)

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

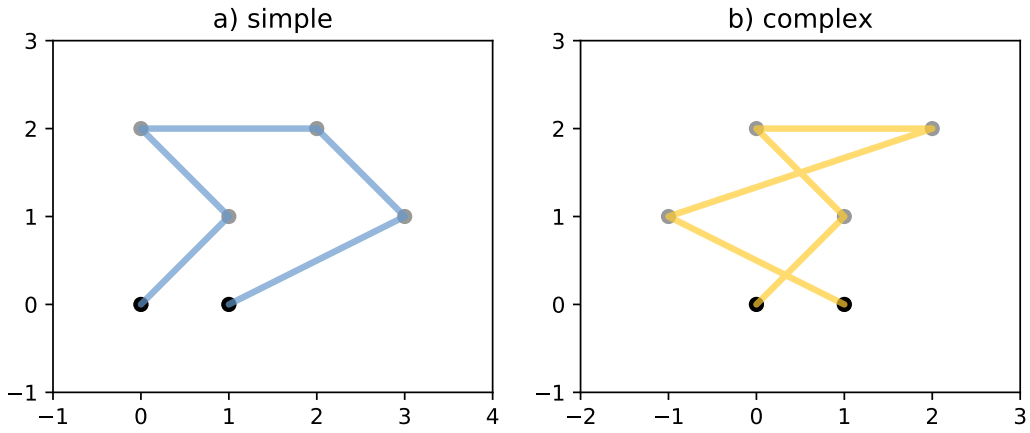
The constructed *LineString* object represents one or more connected linear splines between the points. Repeated points in the ordered sequence are allowed, but may incur performance penalties and should be avoided. A *LineString* may cross itself (*i.e.* be *complex* and not *simple*).

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.geometry import LineString
>>> line = LineString([(0, 0), (1, 1)])
>>> line.area
```

(continues on next page)



(continued from previous page)

```
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*.

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

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

```
>>> LineString(line)
<shapely.geometry.linestring.LineString object at 0x...>
```

A *LineString* may also be constructed using a 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)])
<shapely.geometry.linestring.LineString object at 0x...>
```

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.

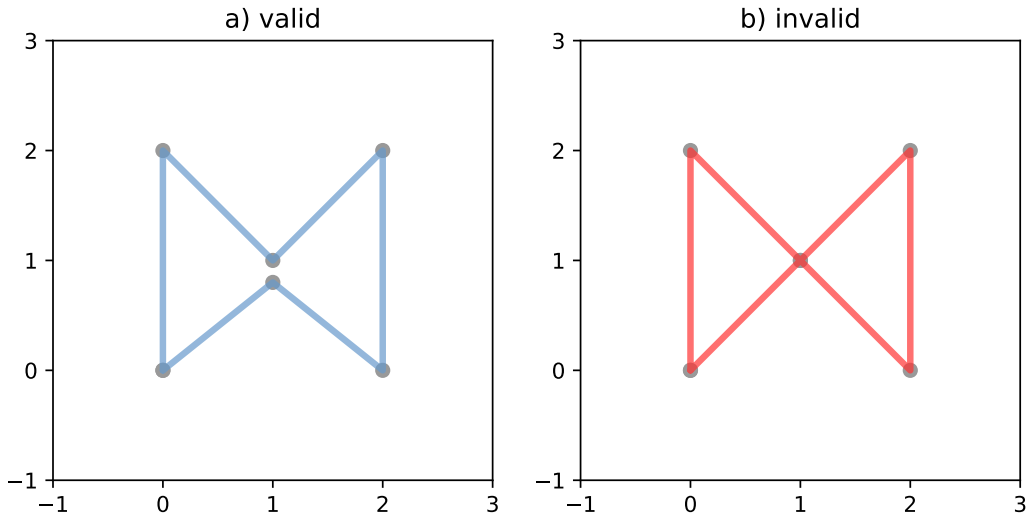


Figure 2. A valid *LinearRing* on the left, an invalid self-touching *LinearRing* on the right. The points that describe the rings are shown in grey. A ring's boundary is *empty*.

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.geometry.polygon import LinearRing
>>> ring = LinearRing([(0, 0), (1, 1), (1, 0)])
>>> ring.area
0.0
>>> ring.length
3.4142135623730949
```

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)
<shapely.geometry.polygon.LinearRing object at 0x...>
```

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.

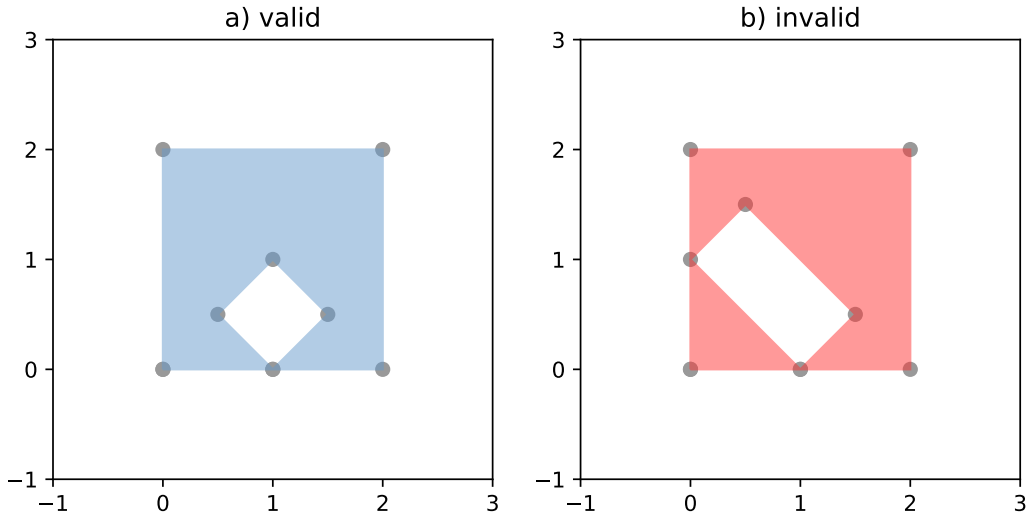


Figure 3. On the left, a valid *Polygon* with one interior ring that touches the exterior ring at one point, and on the right a *Polygon* that is *invalid* because its interior ring touches the exterior ring at more than one point. The points that describe the rings are shown in grey.

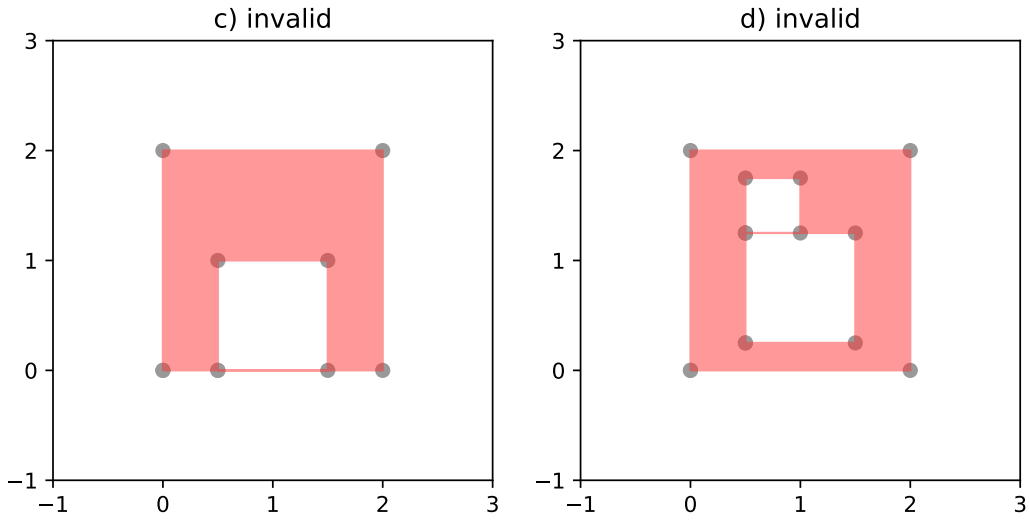


Figure 4. On the left, a *Polygon* that is *invalid* because its exterior and interior rings touch along a line, and on the right, a *Polygon* that is *invalid* because its interior rings touch along a line.

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

```
>>> from shapely.geometry import Polygon
>>> polygon = Polygon([(0, 0), (1, 1), (1, 0)])
>>> polygon.area
0.5
>>> polygon.length
3.4142135623730949
```

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
6.5507620529190334
```

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.geometry import box
>>> b = box(0.0, 0.0, 1.0, 1.0)
>>> b
<shapely.geometry.polygon.Polygon object at 0x...>
>>> 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.

New in version 1.2.10.

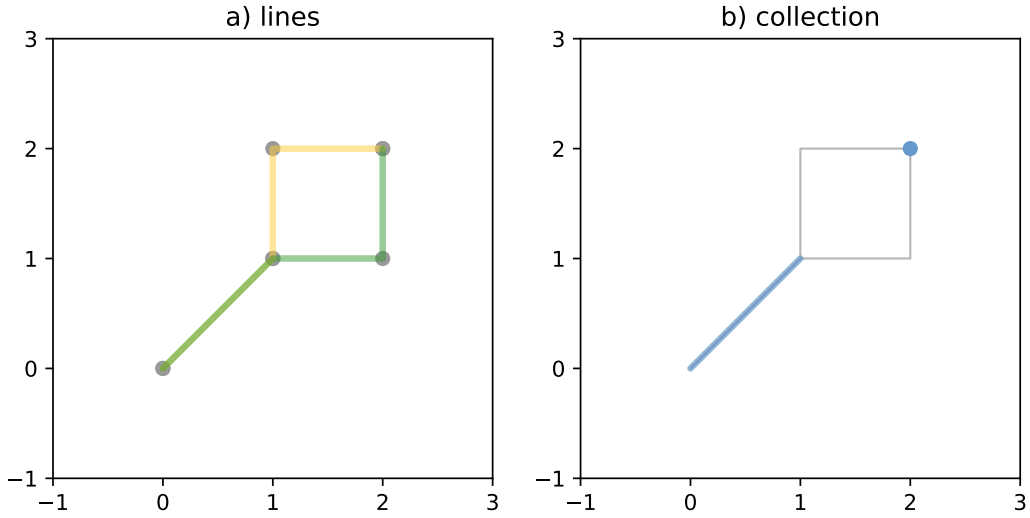
Collections

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
<shapely.geometry.collection.GeometryCollection object at 0x...>
>>> from pprint import pprint
>>> pprint(list(x))
[<shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>]
```

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 or via the iterator protocol using `in` or `list()`.




```
>>> pprint(list(x.geoms))
[<shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>]
>>> pprint(list(x))
[<shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>]
```

Collections can also be sliced.

```
>>> from shapely.geometry import MultiPoint
>>> m = MultiPoint([(0, 0), (1, 1), (1,2), (2,2)])
>>> m[:1].wkt
'MULTIPOINT (0.0000000000000000 0.0000000000000000)'
>>> m[3:].wkt
'MULTIPOINT (2.0000000000000000 2.0000000000000000)'
>>> m[4:].wkt
'GEOMETRYCOLLECTION EMPTY'
```

New in version 1.2.14.

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.geometry 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 or via the iterator protocol using `in` or `list()`.

```
>>> import pprint
>>> pprint.pprint(list(points.geoms))
[<shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.point.Point object at 0x...>]
>>> pprint.pprint(list(points))
[<shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.point.Point object at 0x...>]
```

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

```
>>> MultiPoint([Point(0, 0), Point(1, 1)])
<shapely.geometry.multipoint.MultiPoint object at 0x...>
```

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.

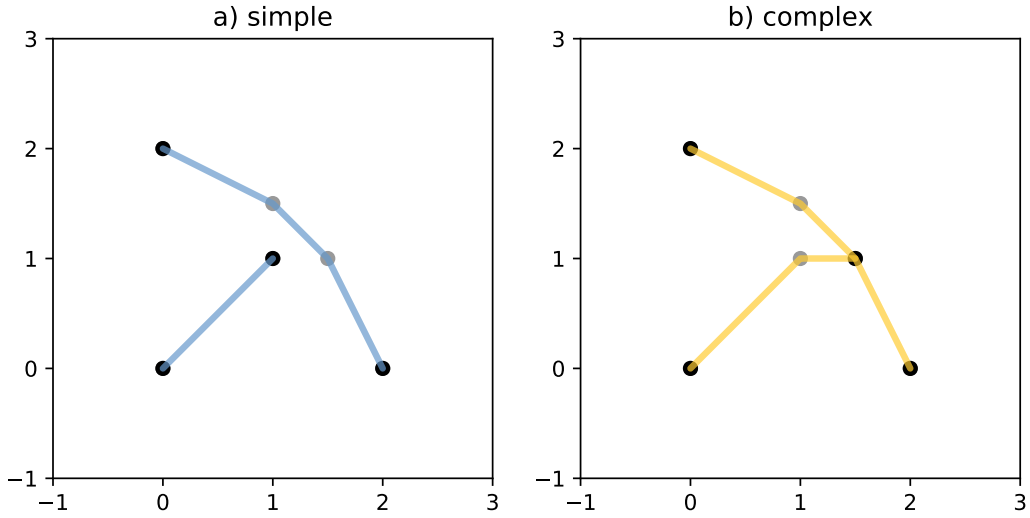
A *MultiLineString* has zero area and non-zero length.

```
>>> from shapely.geometry import MultiLineString
>>> coords = [((0, 0), (1, 1)), ((-1, 0), (1, 0))]
>>> lines = MultiLineString(coords)
>>> lines.area
0.0
>>> lines.length
3.4142135623730949
```

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 or via the iterator protocol using `in` or `list()`.




```
>>> len(lines.geoms)
2
>>> pprint.pprint(list(lines.geoms))
[<shapely.geometry.linestring.LineString object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>]
>>> pprint.pprint(list(lines))
[<shapely.geometry.linestring.LineString object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>]
```

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

```
>>> MultiLineString(lines)
<shapely.geometry.multilinestring.MultiLineString object at 0x...>
>>> MultiLineString(lines.geoms)
<shapely.geometry.multilinestring.MultiLineString object at 0x...>
```

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.

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

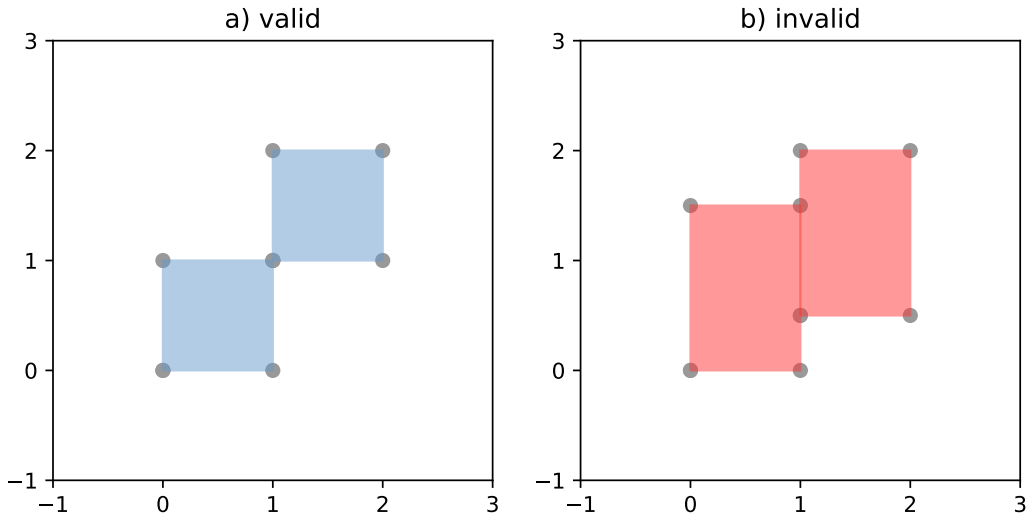


Figure 7. On the left, a *valid MultiPolygon* with 2 members, and on the right, a *MultiPolygon* that is invalid because its members touch at an infinite number of points (along a line).

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 or via the iterator protocol using `in` or `list()`.

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

(continues on next page)

(continued from previous page)

```
()
>>> line.coords
[]
```

The coordinates of a empty feature can be set, after which the geometry is no longer empty.

```
>>> line.coords = [(0, 0), (1, 1)]
>>> line.is_empty
False
>>> line.length
1.4142135623730951
>>> line.bounds
(0.0, 0.0, 1.0, 1.0)
```

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 0x00000276EED1C7F0>
```

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 0x00000276EED1C048>
```

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[2].coords
<shapely.coords.CoordinateSequence object at 0x00000276EFB9B320>
```

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.

Note: Linear referencing methods require GEOS 3.2.0 or later.

`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
<shapely.geometry.point.Point object at 0x740570>
>>> ip.wkt
'POINT (0.5000000000000000 1.0000000000000000)'
>>> LineString([(0, 0), (0, 1), (1, 1)]).interpolate(0.75, normalized=True).wkt
'POINT (0.5000000000000000 1.0000000000000000)'
```

`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)])
>>> pprint([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)]]
>>> pprint([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)]]
```

1.2.3 Predicates and Relationships

Objects of the types explained in *Geometric Objects* provide standard[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
>>> ring.coords = list(ring.coords)[::-1]
>>> ring.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()` or `itertools.ifilter()`.

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

`object.`**`is_ring`**
Returns `True` if the feature is closed. 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[1].

A valid *LinearRing* may not cross itself or touch itself at a single point. 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
```

(continues on next page)

(continued from previous page)

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.almost_equals(other[, decimal=6])`
Returns `True` if the object is approximately equal to the *other* at all points to specified *decimal* place precision.

`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 = filter(line.contains, [Point(), Point(0.5, 0.5)])
>>> len(contained)
1
>>> [p.wkt for p in contained]
['POINT (0.5000000000000000 0.5000000000000000)']
```

`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 objects intersect (see above) but neither contains the other.

`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(object):
    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, and the reason why we use *`within()`* in reverse instead of *`contains()`*. Trying it out on features *d* and *c*, we see that it works.

```
>>> d < c
True
>>> 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[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[4] and http://pypi.python.org/pypi/de9im.

1.2.4 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 (null) collection.

```
>> coords = [((0, 0), (1, 1)), ((-1, 0), (1, 0))]
>>> lines = MultiLineString(coords)
>>> lines.boundary
<shapely.geometry.multipoint.MultiPoint object at 0x...>
>>> pprint(list(lines.boundary))
[<shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.point.Point object at 0x...>,
 <shapely.geometry.point.Point object at 0x...>]
```

(continues on next page)

(continued from previous page)

```
>>> lines.boundary.boundary
<shapely.geometry.collection.GeometryCollection object at 0x...>
>>> lines.boundary.boundary.is_empty
True
```

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
<shapely.geometry.point.Point object at 0x...>
>>> LineString([(0, 0), (1, 1)]).centroid.wkt
'POINT (0.5000000000000000 0.5000000000000000)'
```

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)
<shapely.geometry.polygon.Polygon object at 0x...>
```

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`.

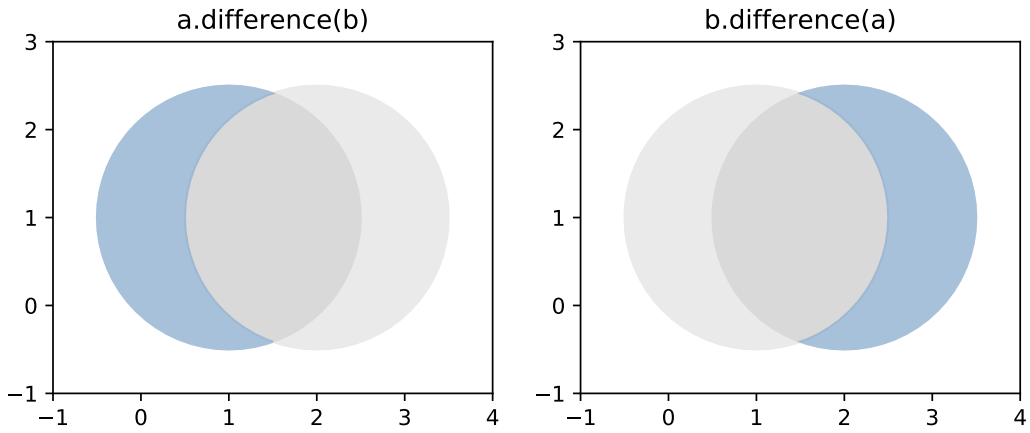
`object.`**`intersection`**`(`*`other`*`)`
Returns a representation of the intersection of this object with the *other* geometric object.

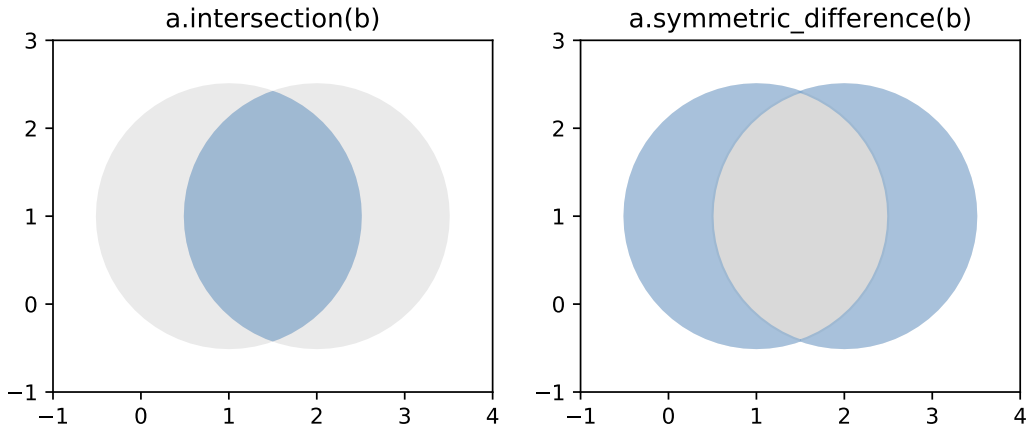
```
>>> a = Point(1, 1).buffer(1.5)
>>> b = Point(2, 1).buffer(1.5)
>>> a.intersection(b)
<shapely.geometry.polygon.Polygon object at 0x...>
```

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)
<shapely.geometry.multipolygon.MultiPolygon object at ...>
```


a.difference(b)
b.difference(a)


a.intersection(b)
a.symmetric_difference(b)

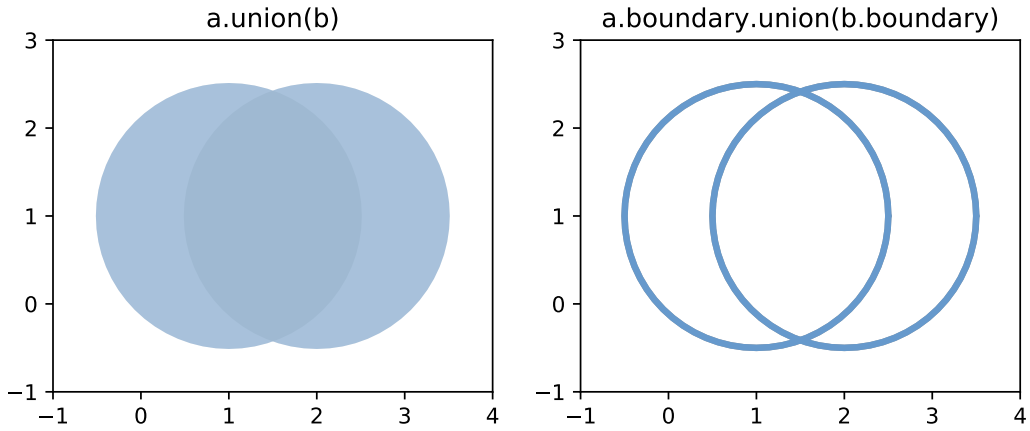
`object.`**`union`**`(other)`
Returns a representation of the union of points from this object and the *other* geometric object.

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)
<shapely.geometry.polygon.Polygon object at 0x...>
```

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
<shapely.geometry.polygon.LinearRing object at 0x...>
>>> a.boundary.union(b.boundary)
<shapely.geometry.multilinestring.MultiLineString object at 0x...>
```



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

Constructive Methods

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

`object.`**`buffer`**`(distance, resolution=16, cap_style=1, join_style=1, mitre_limit=5.0)`
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.geometry.CAP_STYLE* (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.geometry.JOIN_STYLE* (see below).

shapely.geometry.**CAP_STYLE**

Attribute	Value
round	1
flat	2
square	3

shapely.geometry.**JOIN_STYLE**

Attribute	Value
round	1
mitre	2
bevel	3

```
>>> from shapely.geometry import CAP_STYLE, JOIN_STYLE
>>> CAP_STYLE.flat
2
>>> JOIN_STYLE.bevel
3
```

A positive distance has an effect of dilation; a negative distance, erosion. The optional *resolution* 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.

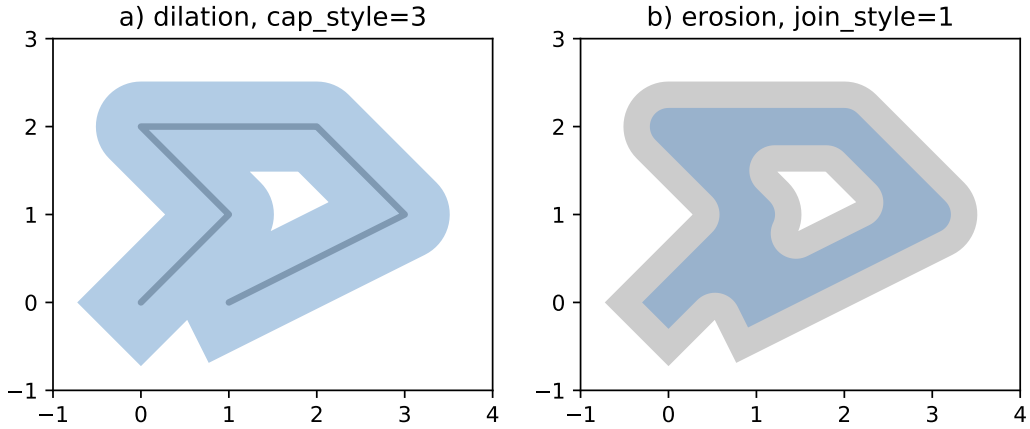
The default (*resolution* of 16) buffer of a point is a polygonal patch with 99.8% of the area of the circular disk it approximates.

```
>>> p = Point(0, 0).buffer(10.0)
>>> len(p.exterior.coords)
66
>>> p.area
313.65484905459385
```

With a *resolution* 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
```

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
<shapely.geometry.multipolygon.MultiPolygon object at ...>
>>> len(clean)
2
>>> list(clean[0].exterior.coords)
[(0.0, 0.0), (0.0, 2.0), (1.0, 1.0), (0.0, 0.0)]
>>> list(clean[1].exterior.coords)
[(1.0, 1.0), (2.0, 2.0), (2.0, 0.0), (1.0, 1.0)]
```

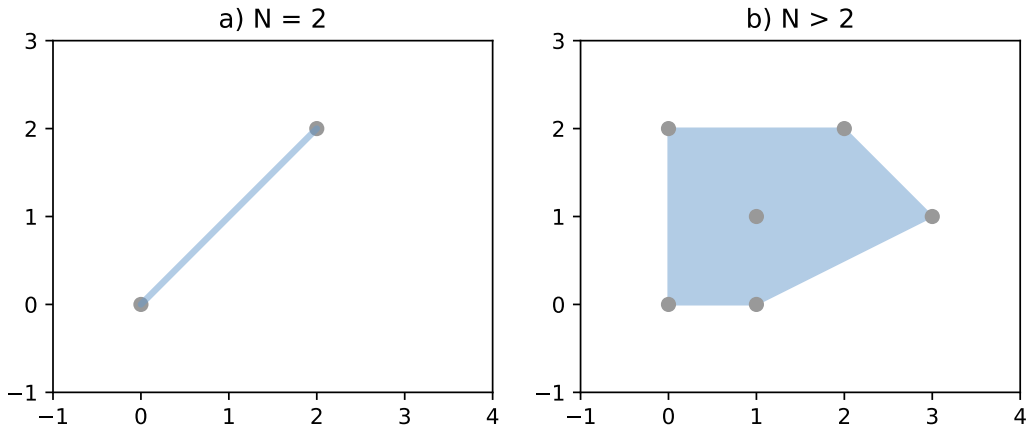
Buffering splits the polygon in two at the point where they touch.

`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
<shapely.geometry.point.Point object at 0x...>
>>> MultiPoint([(0, 0), (1, 1)]).convex_hull
<shapely.geometry.linestring.LineString object at 0x...>
>>> MultiPoint([(0, 0), (1, 1), (1, -1)]).convex_hull
<shapely.geometry.polygon.Polygon object at 0x...>
```

Figure 10. 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
<shapely.geometry.point.Point object at 0x...>
>>> MultiPoint([(0, 0), (1, 1)]).envelope
<shapely.geometry.polygon.Polygon object at 0x...>
```

`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

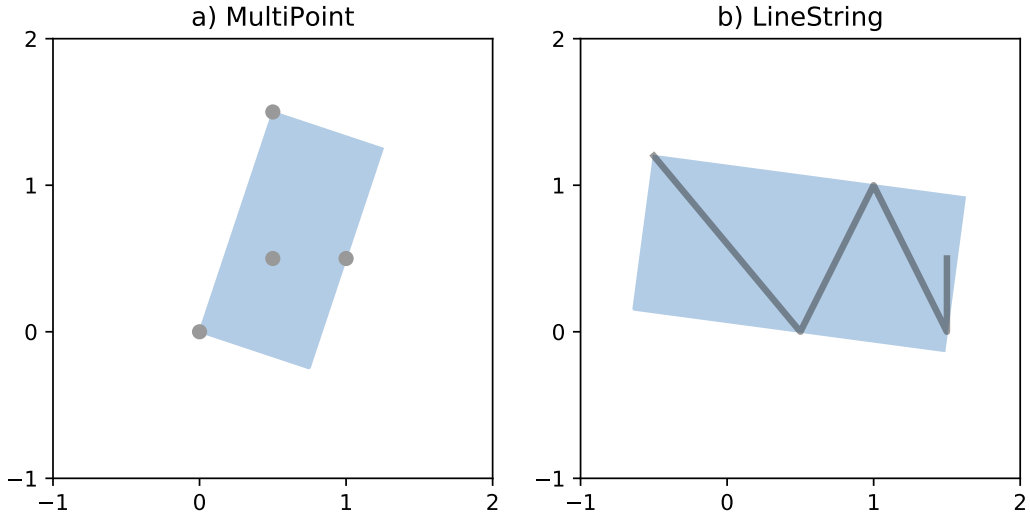
```
>>> Point(0, 0).minimum_rotated_rectangle
<shapely.geometry.point.Point object at 0x...>
>>> MultiPoint([(0,0),(1,1),(2,0.5)]).minimum_rotated_rectangle
<shapely.geometry.polygon.Polygon object at 0x...>
```

Figure 11. 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.

Distance must be a positive float value. The side parameter may be 'left' or 'right'. The resolution of the offset around each vertex of the object is parameterized as in the buffer method.

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



Severely mitered corners can be controlled by the mitre_limit parameter (spelled in British English, en-gb). The ratio of the distance from the corner to the end of the mitred offset corner is the miter ratio. Corners with a ratio which exceed the limit will be beveled.

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

Figure 12. 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 13. Large and small mitre_limit values for left and right offsets.

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

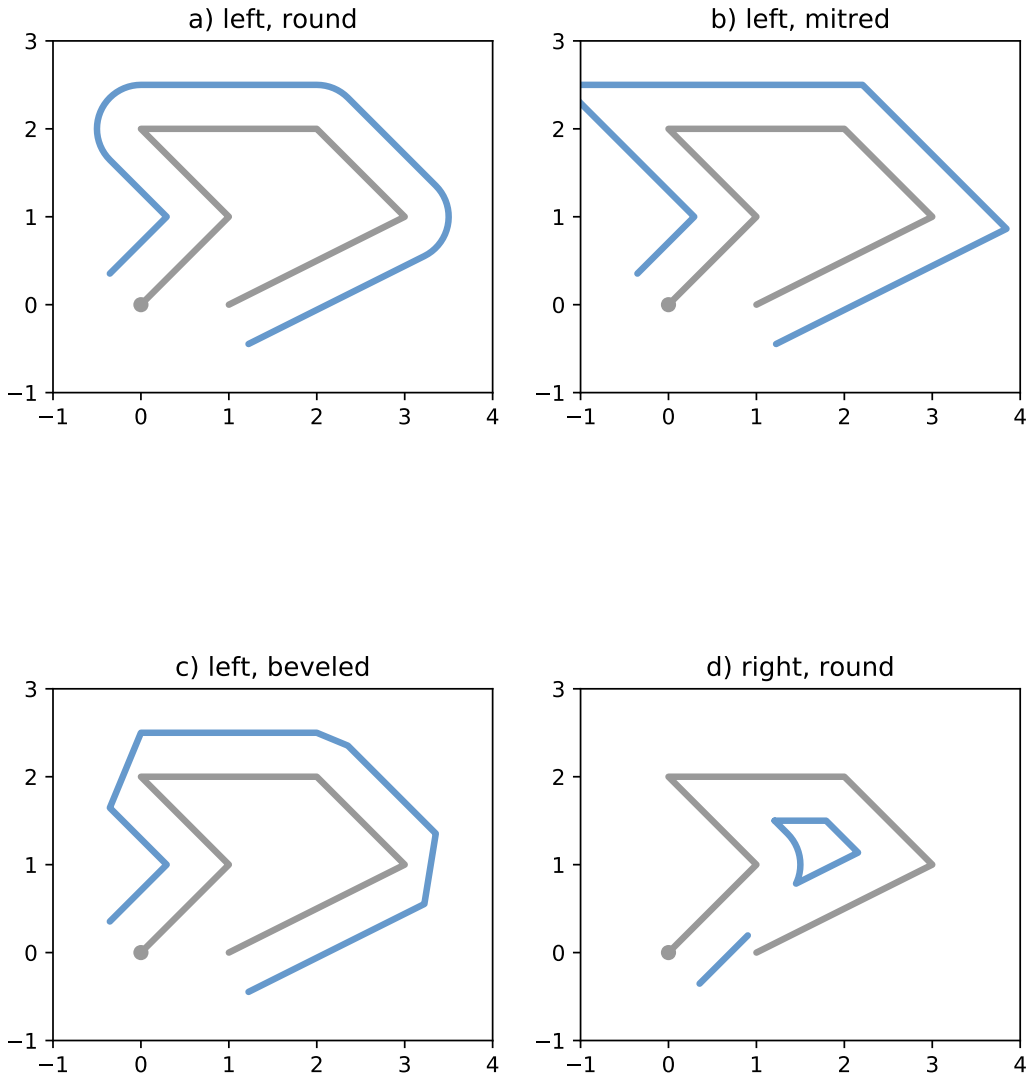
Returns a simplified representation of the geometric object.

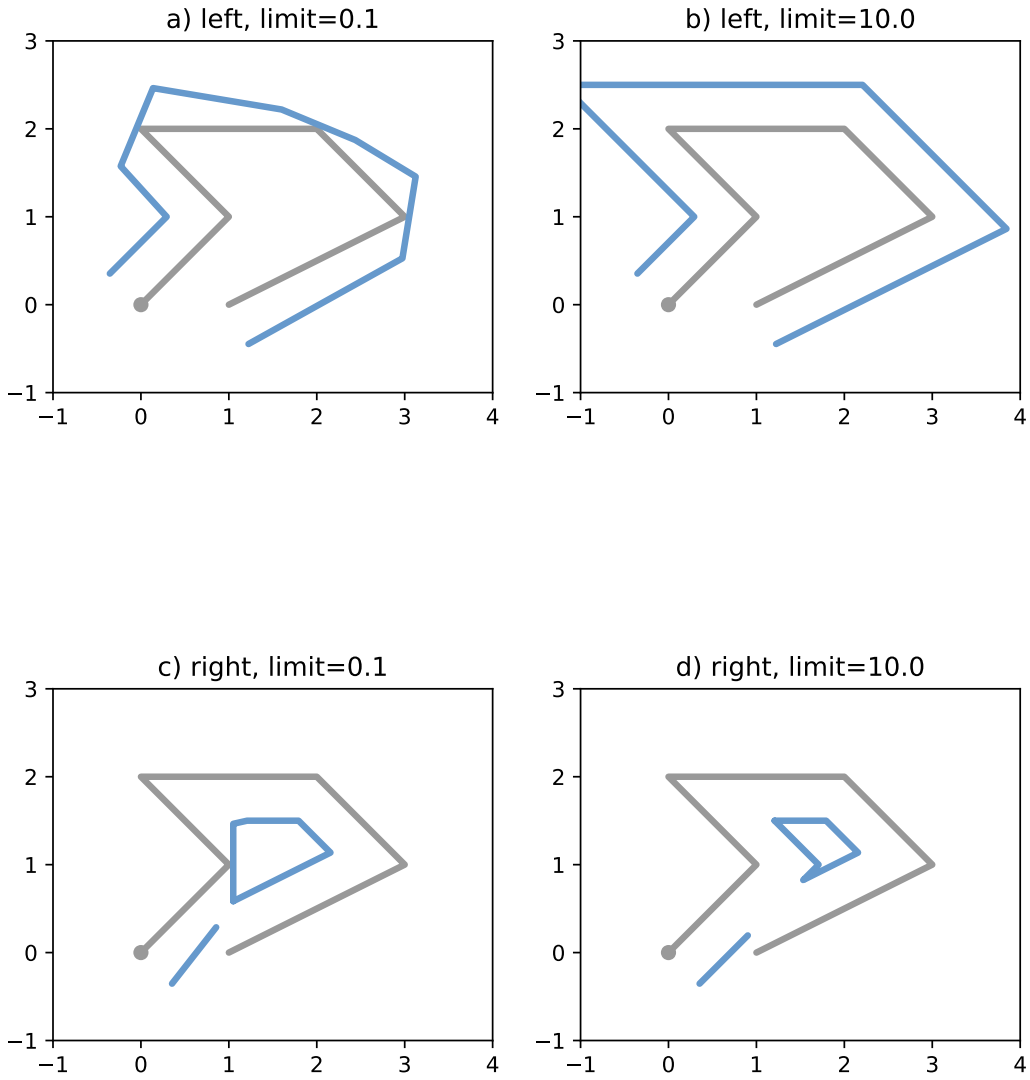
All points in the simplified object will be within the *tolerance* distance of the original geometry. By default a slower algorithm is used that preserves topology. If preserve topology is set to `False` the much quicker Douglas-Peucker algorithm[6] is used.

```
>>> p = Point(0.0, 0.0)
>>> x = p.buffer(1.0)
>>> x.area
3.1365484905459389
>>> len(x.exterior.coords)
66
>>> s = x.simplify(0.05, preserve_topology=False)
```

(continues on next page)

[6] David H. Douglas and Thomas K. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature," Cartographica: The International Journal for Geographic Information and Geovisualization, vol. 10, Dec. 1973, pp. 112-122.


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

(continued from previous page)

```
>>> s.area
3.0614674589207187
>>> len(s.exterior.coords)
17
```

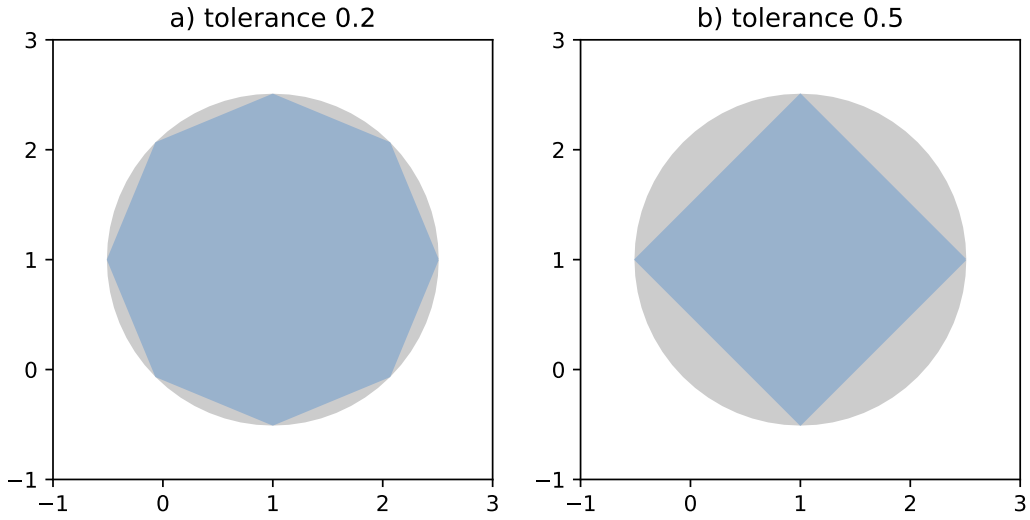


Figure 14. Simplification of a nearly circular polygon using a tolerance of 0.2 (left) and 0.5 (right).

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.

1.2.5 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.

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:

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

or the equations for the transformed coordinates:

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

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:

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

or the equations for the transformed coordinates:

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

`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:

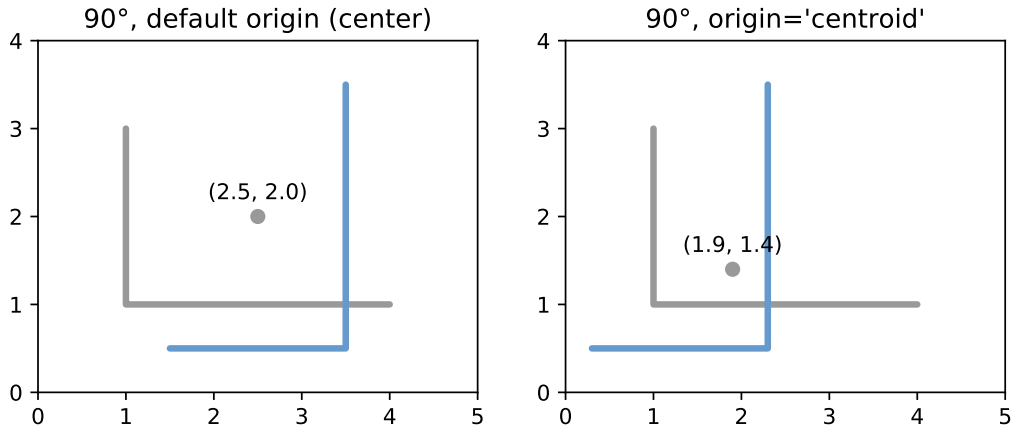
$$\begin{bmatrix} \cos\theta & -\sin\theta & x_{\text{off}} \\ \sin\theta & \cos\theta & y_{\text{off}} \\ 0 & 0 & 1 \end{bmatrix}$$

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

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

```
>>> 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 15. 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:

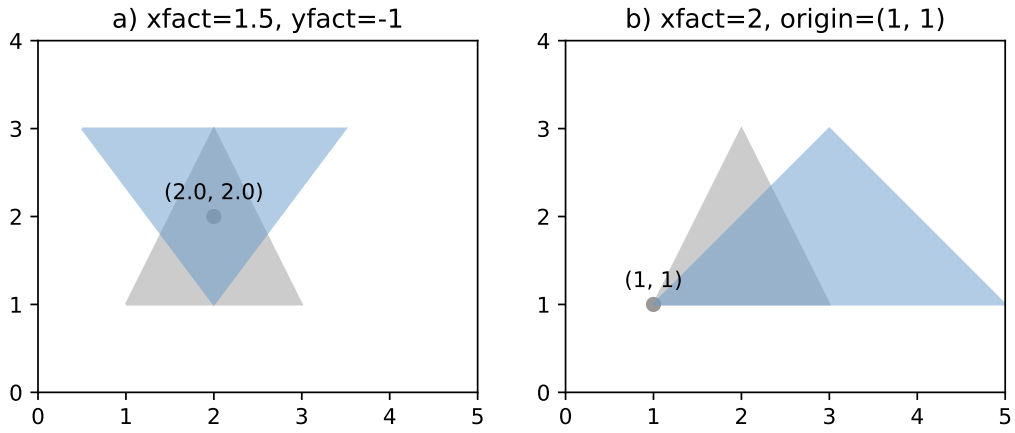
$$\begin{bmatrix} x_{\text{fact}} & 0 & 0 & x_{\text{off}} \\ 0 & y_{\text{fact}} & 0 & y_{\text{off}} \\ 0 & 0 & z_{\text{fact}} & z_{\text{off}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

$$x_{\text{off}} = x_0 - x_0 x_{\text{fact}}$$
$$y_{\text{off}} = y_0 - y_0 y_{\text{fact}}$$
$$z_{\text{off}} = z_0 - z_0 z_{\text{fact}}$$

```
>>> 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 16. 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):

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

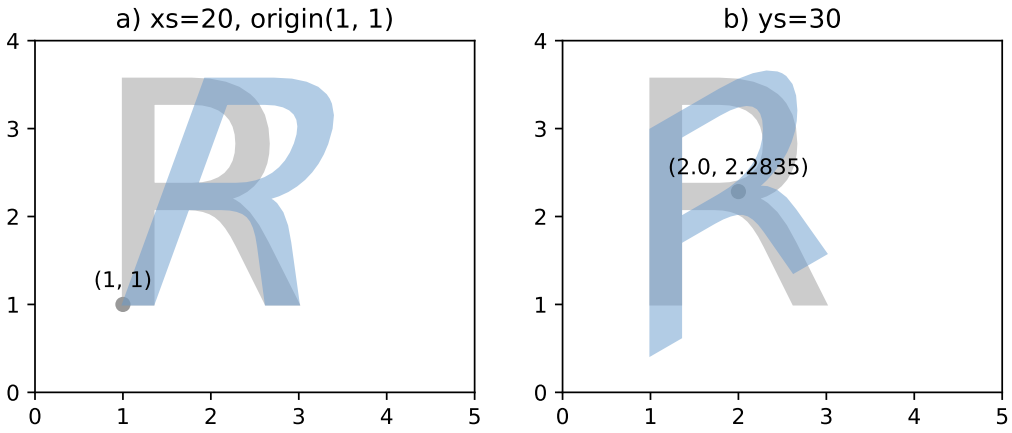
Figure 17. Skewing of a gray "R" to blue result: a) by a shear angle of 20° along the x-direction and an origin at (1, 1); b) by a shear angle of 30° along the y-direction, using default origin.

`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:

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



1.2.6 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 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.

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)
```

A partially applied transform function from pyproj satisfies the requirements for *func*.

```
from shapely.ops import transform
from functools import partial
import pyproj

project = partial(
    pyproj.transform,
    pyproj.Proj(init='epsg:4326'),
    pyproj.Proj(init='epsg:26913'))
```

(continues on next page)

(continued from previous page)

```
g2 = transform(project, g1)
```

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*.

1.2.7 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))
...     ]
>>> pprint(list(polygonize(lines)))
[<shapely.geometry.polygon.Polygon object at 0x...>,
 <shapely.geometry.polygon.Polygon object at 0x...>]
```

`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, dangles, cut edges, 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.

```
>>> 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)),
...     ]
```

(continues on next page)

(continued from previous page)

```
>>> result, dangles, cuts, invalids = polygonize_full(lines)
>>> len(result)
2
>>> list(result.geoms)
[<shapely.geometry.polygon.Polygon object at ...>, <shapely.geometry.polygon.
↪Polygon object at ...>]
>>> list(cuts.geoms)
[<shapely.geometry.linestring.LineString object at ...>, <shapely.geometry.
↪linestring.LineString object at ...>]
```

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

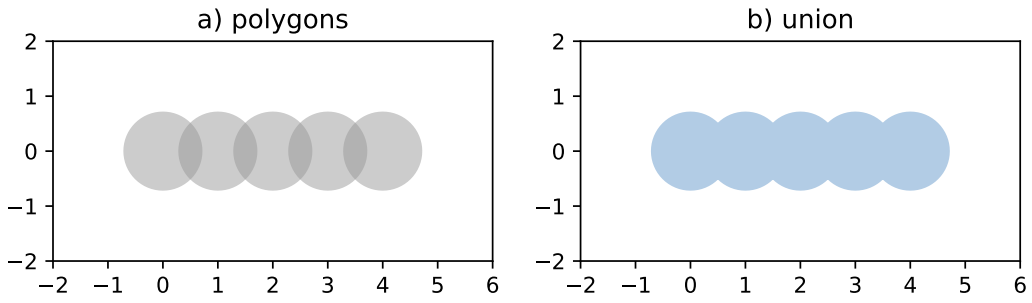
Returns a *LineString* or *MultiLineString* representing the merger of all contiguous elements of *lines*.

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

```
>>> from shapely.ops import linemerge
>>> linemerge(lines)
<shapely.geometry.multilinestring.MultiLineString object at 0x...>
>>> pprint(list(linemerge(lines)))
[<shapely.geometry.linestring.LineString object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>,
 <shapely.geometry.linestring.LineString object at 0x...>]
```

Cascading Unions

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



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

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

```
>>> from shapely.ops import cascaded_union
>>> polygons = [Point(i, 0).buffer(0.7) for i in range(5)]
>>> cascaded_union(polygons)
<shapely.geometry.polygon.Polygon object at 0x...>
```

The function is particularly useful in dissolving *MultiPolygons*.

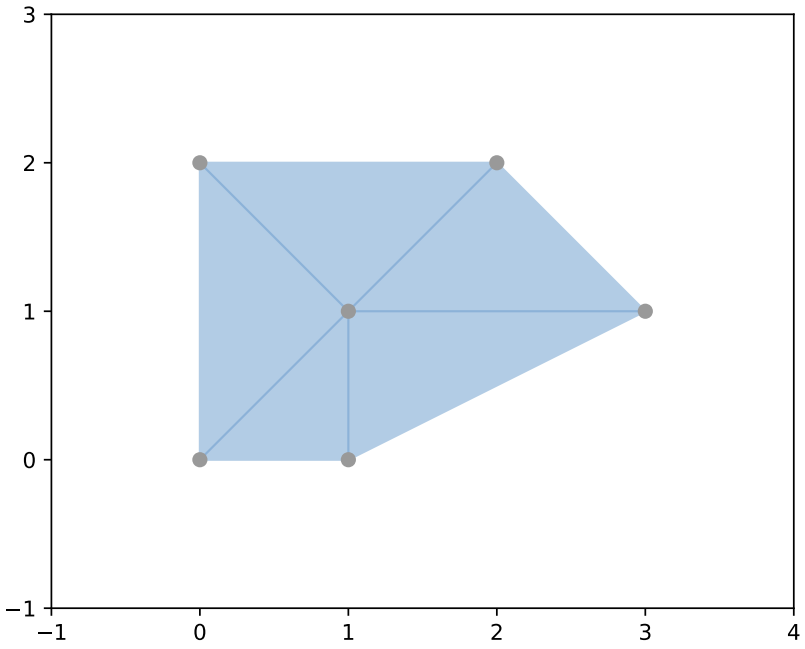
```
>>> m = MultiPolygon(polygons)
>>> m.area
7.6845438018375516
>>> cascaded_union(m).area
6.6103013551167971
```

Note: In 1.2.16 *shapely.ops.cascaded_union()* is superceded by *shapely.ops.unary_union()* if GEOS 3.2+ is used. The unary union function can operate on different geometry types, not only polygons as is the case for the older cascaded unions.

shapely.ops.**unary_union**(*geoms*)
Returns a representation of the union of the given geometric objects.

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 Delaunary 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)])
>>> triangles = triangulate(points)
>>> pprint([triangle.wkt for triangle in triangles])
['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))']
```

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)])
>>> [o.wkt for o in 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.wkt
'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 lineal 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)
>>> forward.wkt
'MULTILINESTRING ((5 0, 10 0))'
>>> backward.wkt
'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

```
>>> pt = Point((1, 1))
>>> line = LineString([(0,0), (2,2)])
>>> result = split(line, pt)
>>> result.wkt
'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 linear geometry.

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

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

New in version 1.7.0

```
>>> line = LineString(([0, 0], [2, 0]))
>>> result = substring(line, 0.5, 0.6)
>>> result.wkt
'LINESTRING (0.5 0, 0.6 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.geometry import Point
>>> 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]'
```

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

```
>>> import shapely
>>> shapely.__version__
'1.3.0'
>>> import shapely.geos
>>> shapely.geos.geos_version
(3, 3, 0)
>>> shapely.geos.geos_version_string
'3.3.0-CAPI-1.7.0'
```

1.2.8 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 an R-tree that you can query with another geometric object.

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

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

These are copied and stored in the R-tree.

New in version 1.4.0.

The *query* method on *STRtree* returns a list of all geometries in the tree that intersect the provided geometry argument. If you want to match geometries of a more specific spatial relationship (eg. crosses, contains, overlaps), consider performing the query on the R-tree, followed by a manual search through the returned subset using the desired binary predicate.

Query-only means that once created, the R-tree is immutable. You cannot add or remove geometries.

```
>>> from shapely.geometry import Point
>>> from shapely.strtree import STRtree
>>> points = [Point(i, i) for i in range(10)]
>>> tree = STRtree(points)
>>> tree.query(Point(2,2).buffer(0.99))
>>> [o.wkt for o in tree.query(Point(2,2).buffer(0.99))]
['POINT (2 2)']
>>> [o.wkt for o in tree.query(Point(2,2).buffer(1.0))]
['POINT (1 1)', 'POINT (2 2)', 'POINT (3 3)']
```

1.2.9 Interoperation

Shapely provides 4 avenues for interoperation with other software.

Well-Known Formats

A *Well Known Text* (WKT) or *Well Known Binary* (WKB) representation[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 (0.0000000000000000 0.0000000000000000)'
>>> Point(0, 0).wkb.encode('hex')
'0101000000000000000000000000000000000000000000'
```

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()`.

`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.wkb import dumps, loads
>>> wkb = dumps(Point(0, 0))
>>> print wkb.encode('hex')
010100000000000000000000000000000000000000
>>> loads(wkb).wkt
'POINT (0.0000000000000000 0.0000000000000000)'
```

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

`shapely.wkt.dumps(ob)`
Returns a WKT representation of *ob*.

`shapely.wkt.loads(wkt)`
Returns a geometric object from a WKT representation *wkt*.

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

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.

```
>>> from numpy import array
>>> array(Point(0, 0))
array([ 0.,  0.])
>>> array(LineString([(0, 0), (1, 1)]))
array([[ 0.,  0.],
       [ 1.,  1.]])
```

The `numpy.asarray()` function does not copy coordinate values – at the price of slower Numpy access to the coordinates of Shapely objects.

Note: The Numpy array interface is provided without a dependency on Numpy itself.

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]))
```

The *shapely.geometry.asShape()* family of functions can be used to wrap Numpy coordinate arrays so that they can then be analyzed using Shapely while maintaining their original storage. A 1 x 2 array can be adapted to a point

```
>>> from shapely.geometry import asPoint
>>> pa = asPoint(array([0.0, 0.0]))
>>> pa.wkt
'POINT (0.0000000000000000 0.0000000000000000)'
```

and a N x 2 array can be adapted to a line string

```
>>> from shapely.geometry import asLineString
>>> la = asLineString(array([[1.0, 2.0], [3.0, 4.0]]))
>>> la.wkt
'LINESTRING (1.0000000000000000 2.0000000000000000, 3.0000000000000000 4.
↪0000000000000000)'
```

Polygon and MultiPoint can also be created from N x 2 arrays:

```
>>> from shapely.geometry import asMultiPoint
>>> ma = asMultiPoint(np.array([[1.1, 2.2], [3.3, 4.4], [5.5, 6.6]]))
>>> ma.wkt
'MULTIPOINT (1.1 2.2, 3.3 4.4, 5.5 6.6)'

>>> from shapely.geometry import asPolygon
>>> pa = asPolygon(np.array([[1.1, 2.2], [3.3, 4.4], [5.5, 6.6]]))
>>> pa.wkt
'POLYGON ((1.1 2.2, 3.3 4.4, 5.5 6.6, 1.1 2.2))'
```

Python Geo Interface

Any object that provides the GeoJSON-like Python geo interface can be adapted and used as a Shapely geometry using the *shapely.geometry.asShape()* or *shapely.geometry.shape()* functions.

shapely.geometry.**asShape**(*context*)
Adapts the context to a geometry interface. The coordinates remain stored in the context.

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(object):
...     def __init__(self, d):
...         self.__geo_interface__ = d
>>> thing = GeoThing({"type": "Point", "coordinates": (0.0, 0.0)})
>>> geom = shape(thing)
```

(continues on next page)

(continued from previous page)

```
>>> 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 new, independent geometry with coordinates *copied* from the context.

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)}
```

1.2.10 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.

New in version 1.2.10.

The `shapely.speedups` module contains performance enhancements written in C. They are automatically installed when Python has access to a compiler and GEOS development headers during installation.

You can check if the speedups are installed with the `available` attribute. To enable the speedups call `enable()`. You can revert to the default implementation with `disable()`.

```
>>> from shapely import speedups
>>> speedups.available
True
>>> speedups.enable()
```

New in version 1.6.0.

Speedups are now enabled by default if they are available. You can check if speedups are enabled with the `enabled` attribute.

```
>>> from shapely import speedups
>>> speedups.enabled
True
```

1.2.11 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/Toblerity/Shapely/tree/master/docs/.

1.2.12 References

CHAPTER 2

Indices and tables

- genindex
- search

Index

Symbols

__eq__() (object method), 35

A

almost_equals() (object method), 35
area (object attribute), 20

B

boundary (object attribute), 38
bounds (object attribute), 20
buffer() (object method), 41

C

centroid (object attribute), 39
contains() (object method), 35
convex_hull (object attribute), 43
crosses() (object method), 36

D

difference() (object method), 39
disjoint() (object method), 36
distance() (object method), 20

E

envelope (object attribute), 43
equals() (object method), 35

G

geom_type (object attribute), 20

H

has_z (object attribute), 33
hausdorff_distance() (object method), 20

I

interpolate() (object method), 32
intersection() (object method), 39
intersects() (object method), 36
is_ccw (object attribute), 33
is_empty (object attribute), 33
is_ring (object attribute), 33
is_simple (object attribute), 34
is_valid (object attribute), 34

L

length (object attribute), 20
LinearRing (built-in class), 23
LineString (built-in class), 21

M

minimum_rotated_rectangle (object attribute), 44
MultiLineString (built-in class), 28
MultiPoint (built-in class), 27
MultiPolygon (built-in class), 29

O

overlaps() (object method), 36

P

parallel_offset() (object method), 44
Point (built-in class), 21
Polygon (built-in class), 24
prepared.prep() (built-in function), 58
project() (object method), 32

R

relate() (object method), 37
relate_pattern() (object method), 37
representative_point() (object method), 20

S

shapely.affinity.affine_transform() (built-in function), 48
shapely.affinity.rotate() (built-in function), 49
shapely.affinity.scale() (built-in function), 49
shapely.affinity.skew() (built-in function), 51
shapely.affinity.translate() (built-in function), 51
shapely.geometry.asShape() (built-in function), 61

shapely.geometry.box() (built-in function), 26
shapely.geometry.CAP_STYLE (built-in variable), 42
shapely.geometry.JOIN_STYLE (built-in variable), 42
shapely.geometry.mapping() (built-in function), 62
shapely.geometry.polygon.orient() (built-in function), 26
shapely.geometry.shape() (built-in function), 61
shapely.ops.cascaded_union() (built-in function), 54
shapely.ops.linemerge() (built-in function), 54
shapely.ops.nearest_points() (built-in function), 56
shapely.ops.polygonize() (built-in function), 53
shapely.ops.polygonize_full() (built-in function), 53
shapely.ops.shared_paths() (built-in function), 57
shapely.ops.snap() (built-in function), 56
shapely.ops.split() (built-in function), 57
shapely.ops.substring() (built-in function), 57
shapely.ops.transform() (built-in function), 52
shapely.ops.triangulate() (built-in function), 55
shapely.ops.unary_union() (built-in function), 55
shapely.wkb.dumps() (built-in function), 60
shapely.wkb.loads() (built-in function), 60
shapely.wkt.dumps() (built-in function), 60
shapely.wkt.loads() (built-in function), 60
simplify() (object method), 45
strtree.STRtree (built-in class), 59
symmetric_difference() (object method), 39

T

touches() (object method), 36

U

union() (object method), 39

W

within() (object method), 36