Module 61-12: Option GIS-Python

Spatial models:

Working with geometries

hes·so business.

Jean-Paul Calbimonte

School of Management

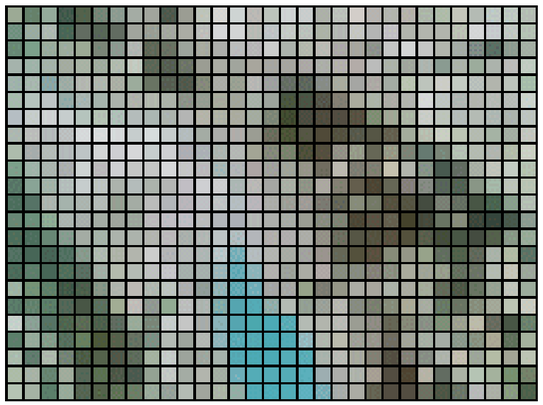
Bachelor of Science HES-SO (BSc) in Business Information Technology

> Representing spatial data

- Simplifications of the real world
- Represented with vector or raster data model
- Other models and extensions: spatio-temporal data model, topological data model

H. Tenkanen, V Heikinheimo, D. Whipp. Python for Geographic Data Analysis https://pythongis.org/

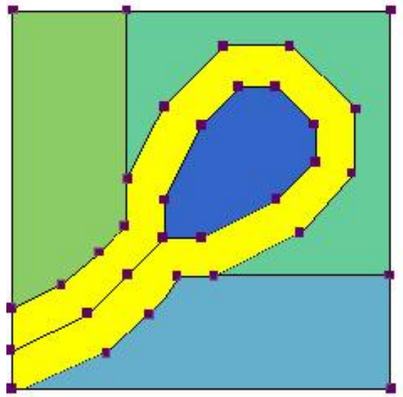
> Vector and Raster

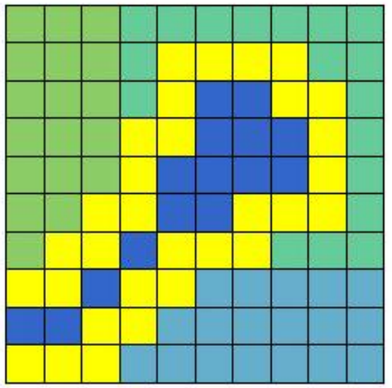


vectorize

rasterize

real world



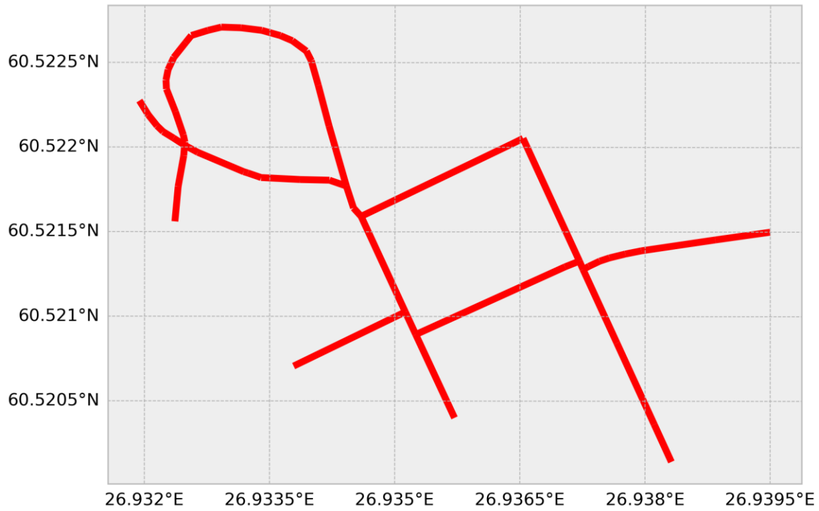


H. Tenkanen, V Heikinheimo, D. Whipp. Python for Geographic Data Analysis https://pythongis.org/

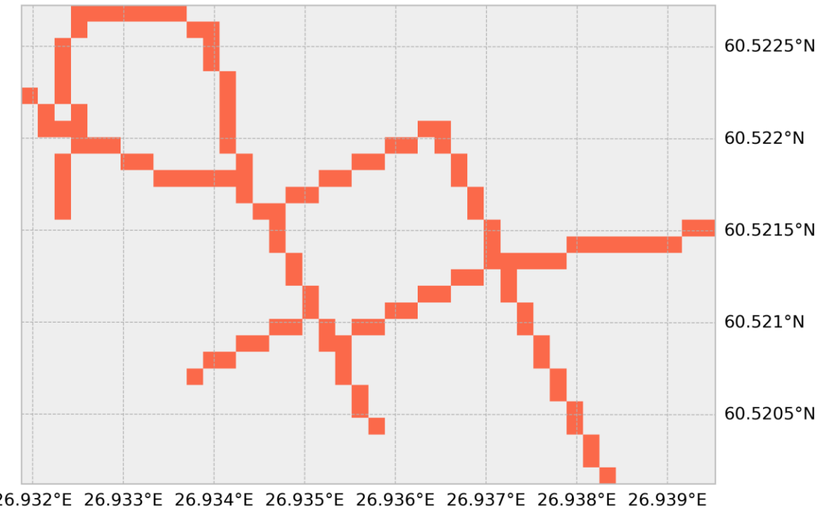
> Vector and Raster

Roads

Vector

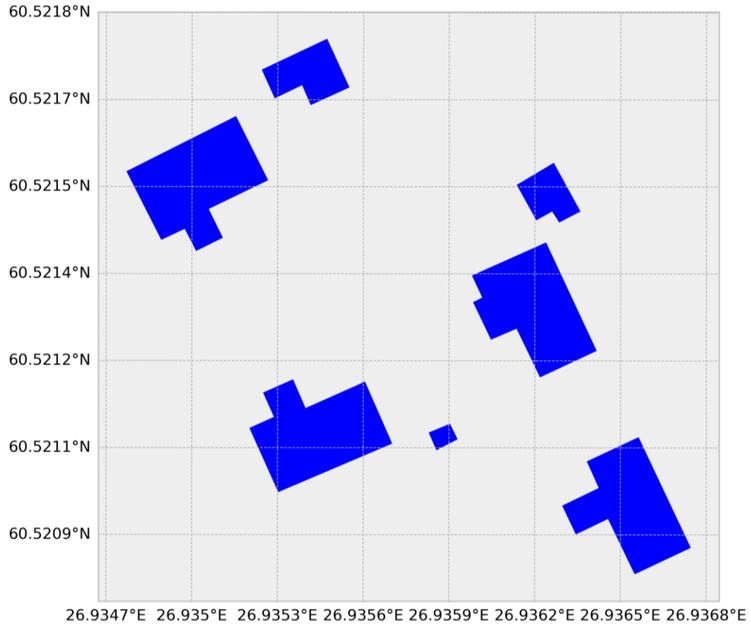


Raster (10 m resolution)

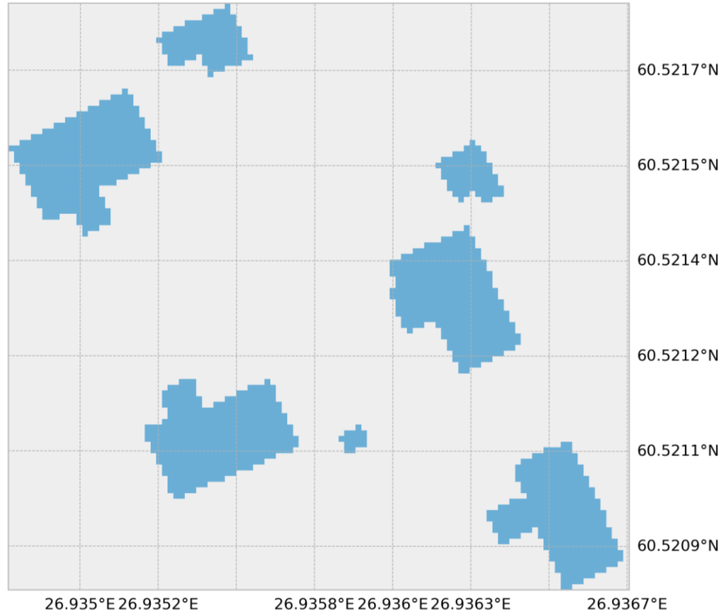


Buildings

Vector

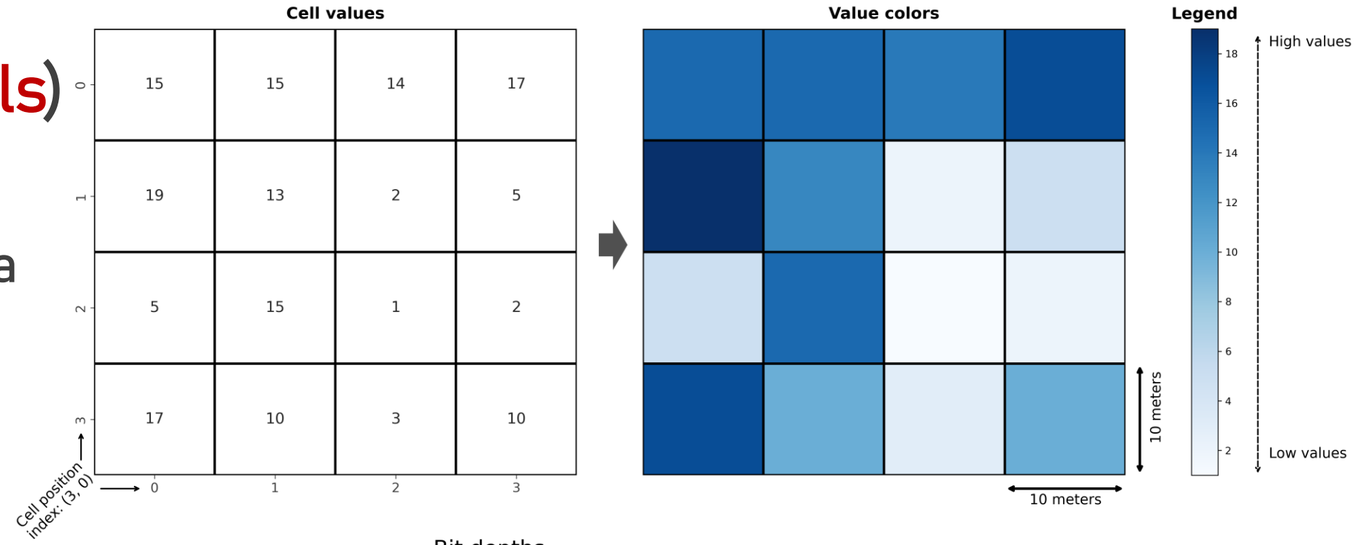


Raster (1 m resolution)



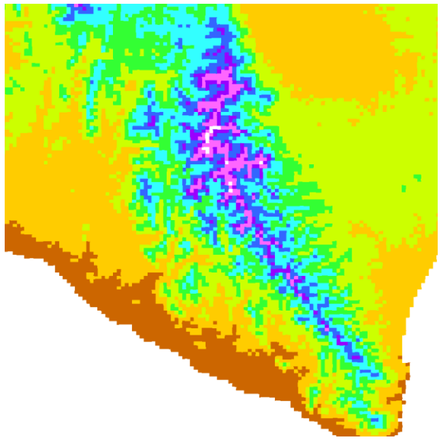
> Raster

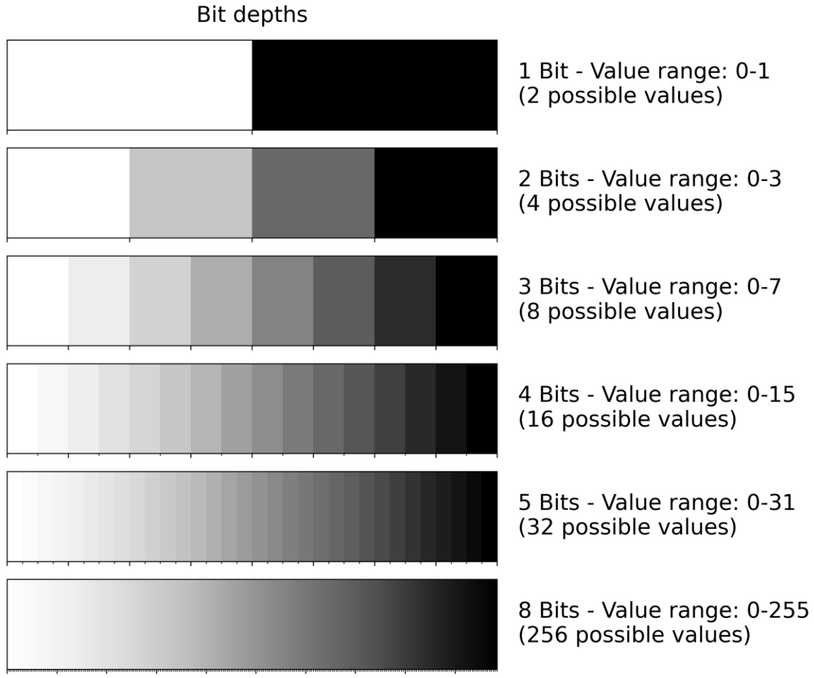
- Data represented as arrays of cells (pixels)



- represent real-world objects, continuous phenomena
- Example: photographs using RGB colors

- Other information: e.g., elevation or temperature



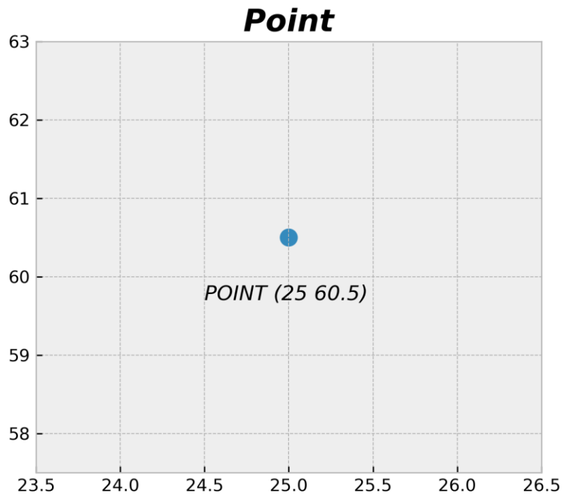


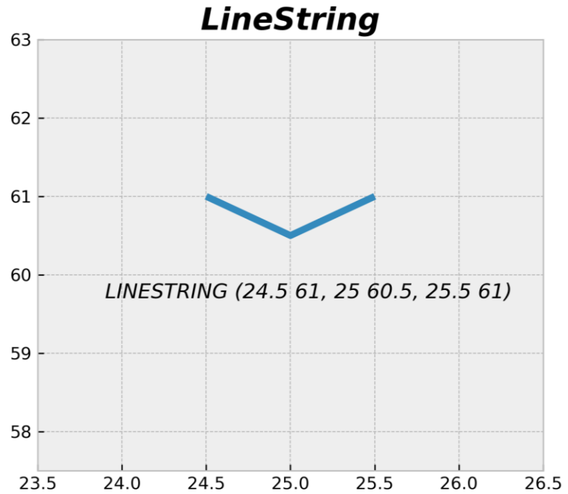
> Raster files

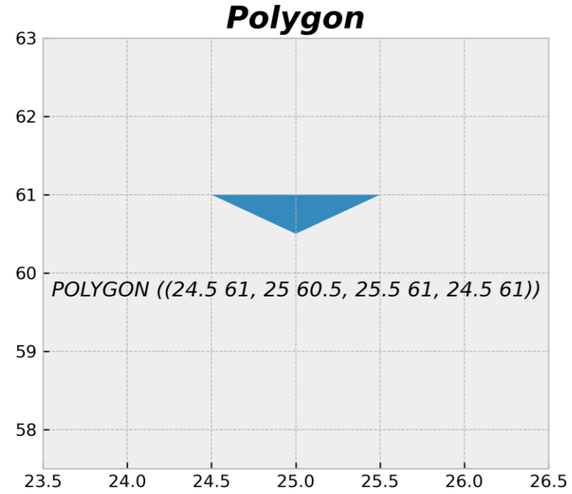
- GeoTIFF: open, non-proprietary raster data format based on TIFF.
- COG: Cloud Optimized GeoTIFF (COG) based on GeoTIFF.
- NetCDF: Network Common Data Form (NetCDF): portable, self-describing and scalable file format for storing array-oriented multidimensional scientific data.
- ASCII Grid: The ASCII Raster File: simple format to transfer raster data between various applications.
- IMG: ERDAS Imagine file format (IMG): proprietary file format accompanied with an .xml metadata file

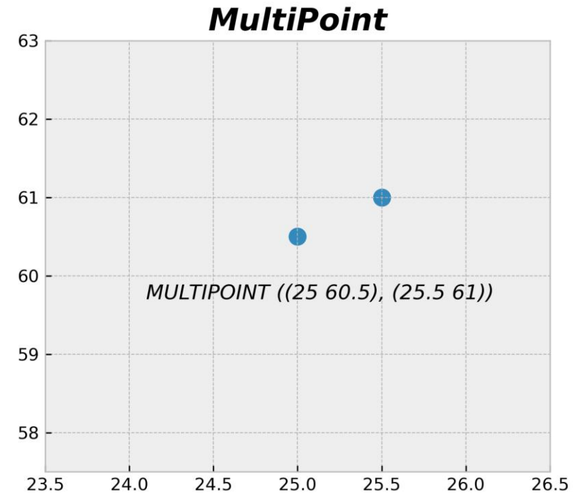
> Vector

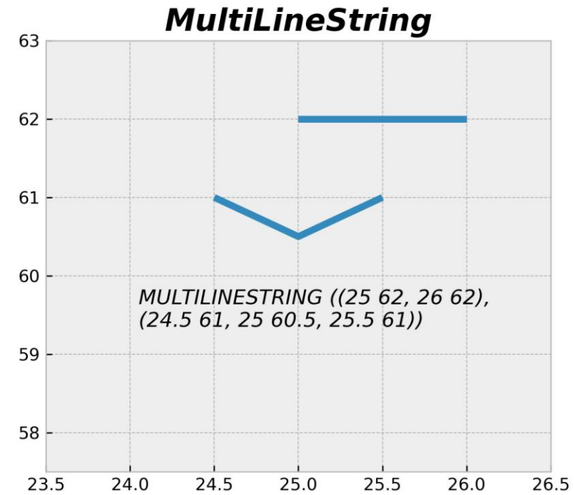
- Fundamental geometric objects: points, lines and areas.
- Defined in Simple Features Access Specification (Herring, 2011)
- Standard (ISO 19125-1) formalized by the Open Geospatial Consortium
- Followed by most programming languages
- Well-known text (WKT) representation.

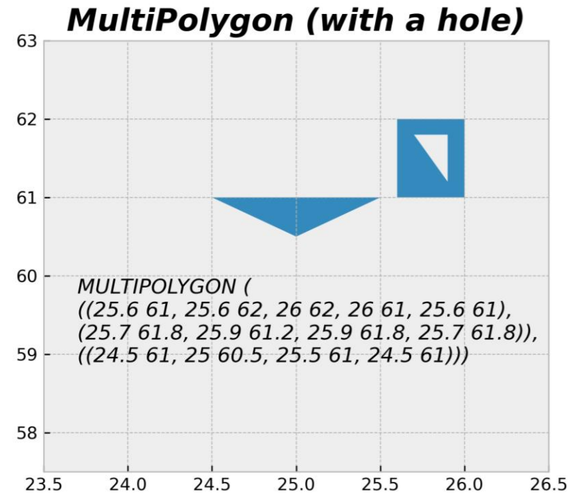












> Vector formats

- Shapefile: introduced by ESRI. Multiple separate files:..shp for geometries, .shx for positional index, .dbf for attribute information + others

- GeoJSON: open standard format extends JSON. Human readable & not compressed. TopoJSON variant.

- GeoPackage: (GPKG): open, non-proprietary, platform-independent, portable and standards-based data format for storing spatial data uses a SQLite database.

- GML: Geography Markup Language: XML based data format defined by the Open Geospatial Consortium (OGC) to express geographical features.

> GeoJSON


```
{
 "type": "FeatureCollection",
  "features": [
     {  "type": "Feature",
       "properties": {"id": 75553155,
                      "timestamp": 1494181812},
       "geometry": {
            "type": "MultiLineString",
            "coordinates": [[[26.938, 60.520], [26.938, 60.520]],
                            [[26.937, 60.521], [26.937, 60.521]],
                            [[26.937, 60.521], [26.936, 60.522]]]
       }
     },
     {  "type": "Feature",
       "properties": {"id": 424099695,
                      "timestamp": 1465572910},
       "geometry": {
            "type": "Polygon",
            "coordinates": [[[26.935, 60.521], [26.935, 60.521], [26.935, 60.521], [26.935, 60.521], [26.935, 60.521]]]
       }
     }
  ]
}
```

> Spatial Model

Shapely library

- set-theoretic analysis
- manipulation of planar features
- PostGIS type geometry operations out of RDBMS
- Python idioms

Types of objects	Shapely classes
• *Point*	`Point`
• *Curve*	`LineString` `LinearRing`
• *Surface*	`Polygon`

• *Collections*	`MultiPoint`
	`MultiLineString`
	`MultiPolygon`

Disclaimer:
the following content is simplified. Full details can be found in the docs of shapely:
https://shapely.readthedocs.io

> Spatial Model

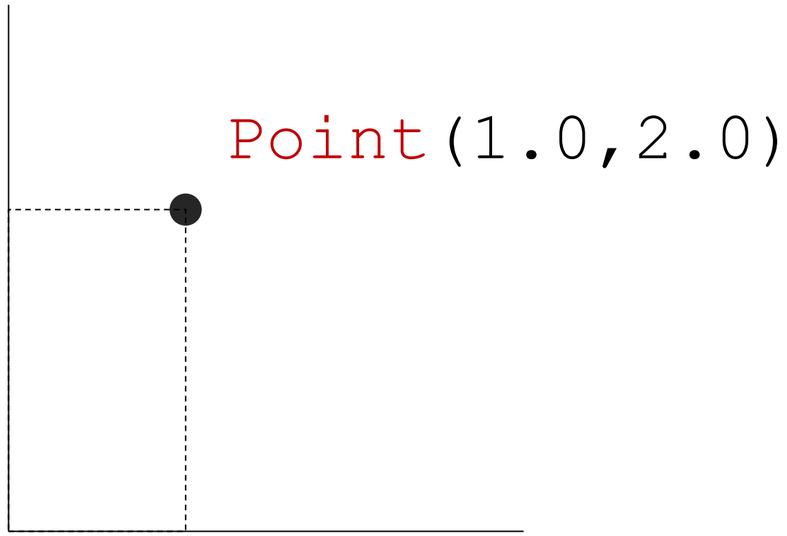
General attributes/methods

- `object.area`: Area (float) of the object.
- `object.bounds`: Tuple (float values) that bounds the object.
- `object.length`: Length (float) of the object.
- `object.geom_type`: String specifying the `GeometryType`
- `object.distance(other)`: Minimum distance to the *`other`* object.
- `object.hausdorff_distance(other)`: Hausdorff distance (float) to the *`other`* object.
- `object.representative_point`: Cheaply computed point guaranteed to be within the geometric object.

> Point Class

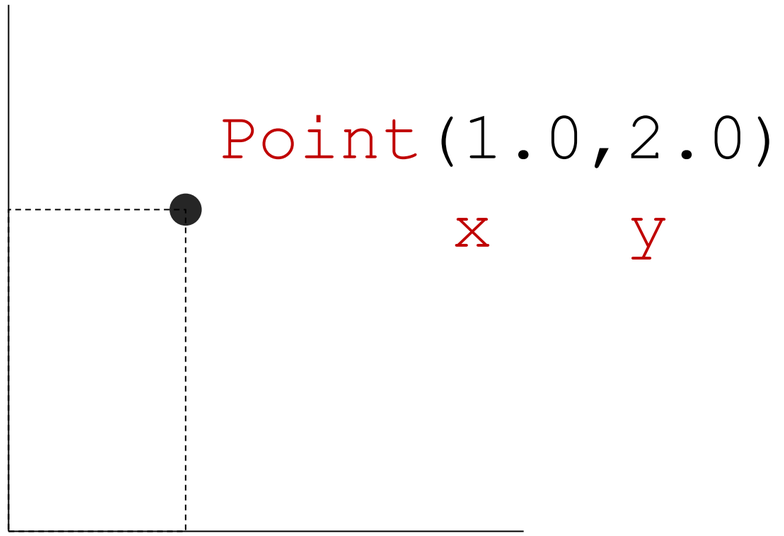
An object that represents a single point in space.
Points are: two-dimensional (x, y) or three dimensional (x, y, z).

```
from shapely.geometry import Point
point = Point(1.0, 2.0)

point.area
0.0

point.length
0.0

point.bounds
(1.0, 2.0, 1.0, 2.0)
```



> Point Class

```
list(point.coords)
[(1.0, 2.0)]

point.x
1.0

point.y
2.0

p2=Point(point)
```



> WKT

A *Well Known Text* (WKT) of any geometric object

```
Point(0, 0).wkt
'POINT (0.0000000000000000 0.0000000000000000)'
```

`shapely.wkt.dumps(ob)`

Returns a WKT representation of *ob*.

`shapely.wkt.loads(wkt)`

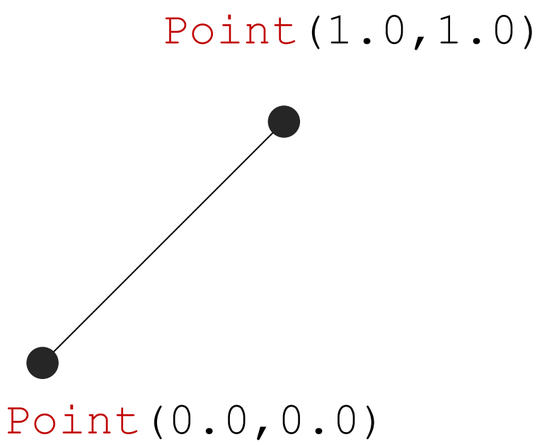
Returns a geometric object from a WKT representation ***wkt***.

```
wkt = dumps(Point(0, 0))
print wkt
POINT (0.0000000000000000 0.0000000000000000)

loads(wkt).wkt
'POINT (0.0000000000000000 0.0000000000000000)'
```

> LineString

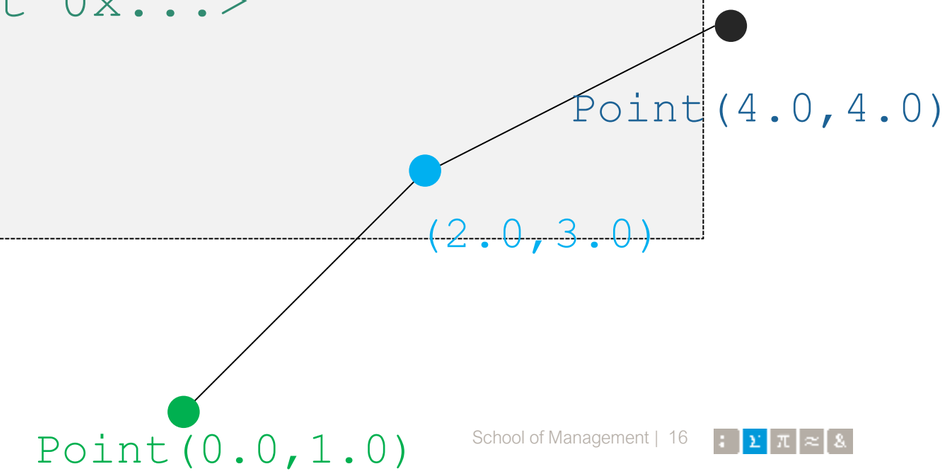
LineString: an object that represents a sequence of points joined together to form a line. Consists of a list of at least two coordinate tuples.

```
from shapely.geometry import LineString
line = LineString([(0, 0), (1, 1)])

line.area
0.0

line.length
1.4142135623730951

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



> LineString

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

line2=LineString(line)
<shapely.geometry.linestring.LineString object at 0x...>
list(line2.coords)
[(0.0, 0.0), (1.0, 1.0)]

line3=LineString([Point(0.0, 1.0), (2.0, 3.0), Point(4.0, 4.0)])
<shapely.geometry.linestring.LineString object at 0x...>

line3.wkt
'LINESTRING (0 1, 2 3, 4 4)'
```

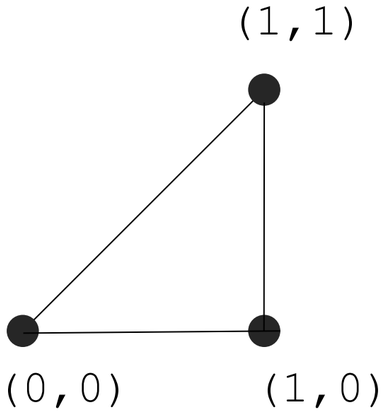


> LinearRing

The *LinearRing* constructor takes an ordered sequence of point tuples.

The sequence may be explicitly closed by passing identical values in the first & last indices.

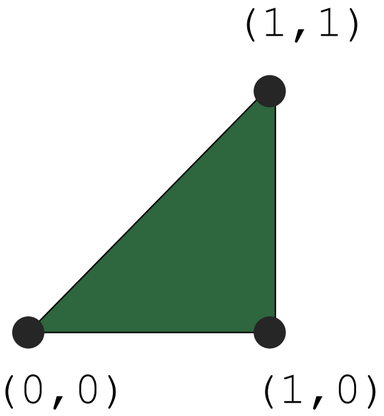
```
from shapely.geometry.polygon import LinearRing
ring = LinearRing([(0, 0), (1, 1), (1, 0)])
ring.area
0.0
ring.length
3.4142135623730949
ring.bounds
(0.0, 0.0, 1.0, 1.0)

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



> Polygon

A Polygon object represents a filled area that consists of a list of at least three coordinate tuples that forms the outerior ring and a (possible) list of hole polygons.

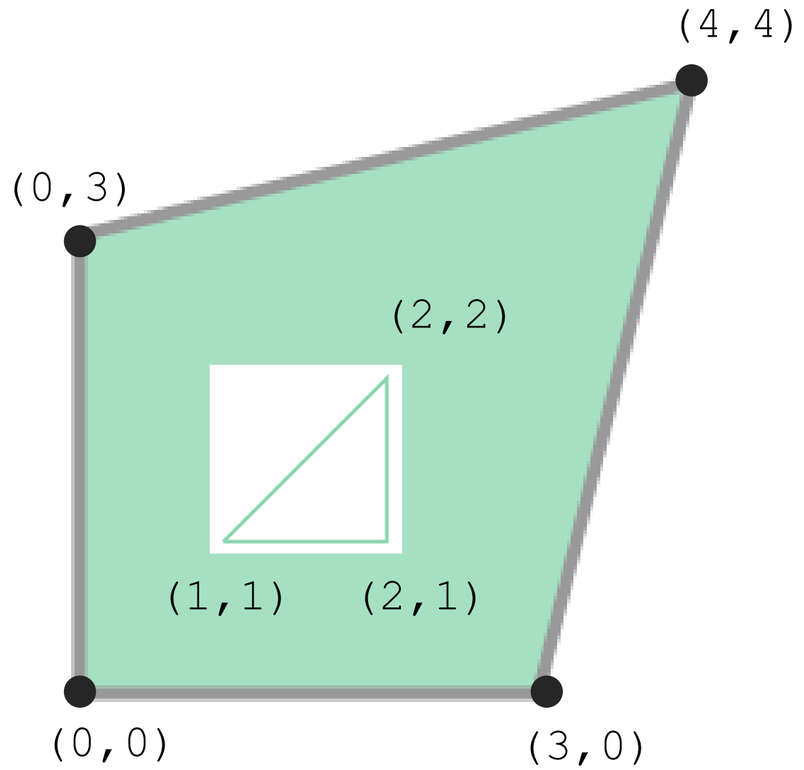
```
from shapely.geometry import Polygon
polygon = Polygon([(0, 0), (1, 1), (1, 0)])
polygon.area
0.5
polygon.length
3.414213562373094 9
polygon.bounds
(0.0, 0.0, 1.0, 1.0)

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



> Polygon

The Polygon constructor also accepts instances of LineString and LinearRing.

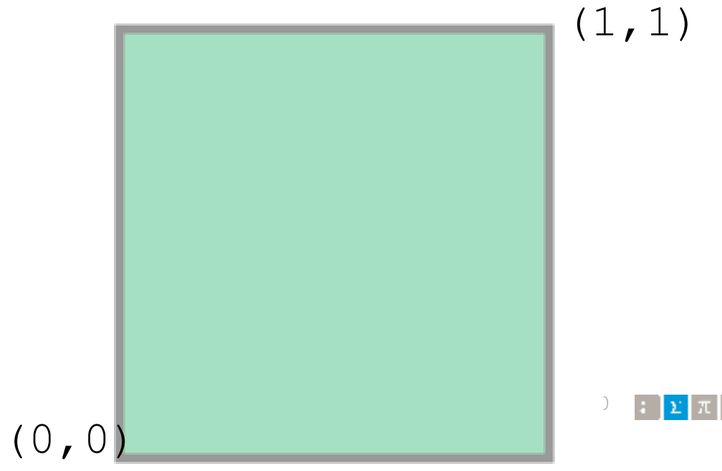
```
coords = [(1, 1), (2, 2), (2, 1)]
r = LinearRing(coords)
s =Polygon(r)
s.area
0.5

ext=[(0,0),(0,3),(4,4),(3,0)]
t = Polygon(ext, [r])

t.area
6.550762052919033 4
```



> Polygon

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

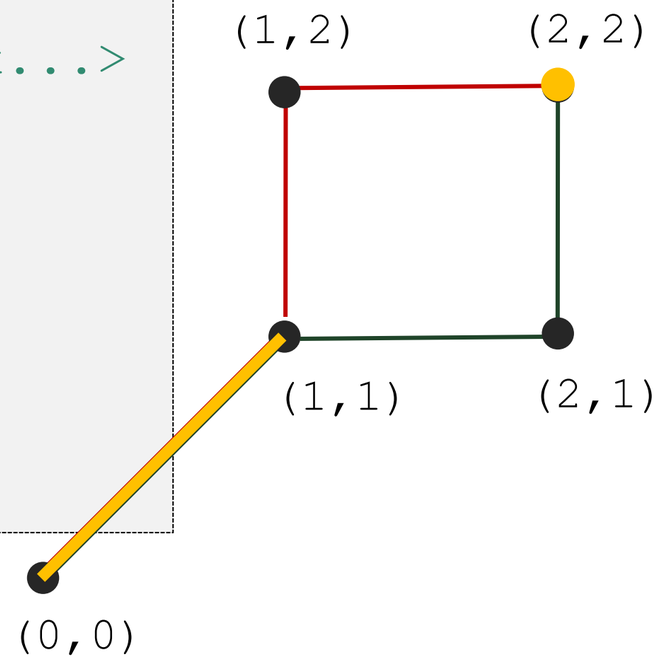
```
from shapely.geometry import box
b = box(0.0, 0.0, 1.0, 1.0)
b
<shapely.geometry.polygon.Polygon object at 0x...>

list(b.exterior.coords)
[(1.0, 0.0), (1.0, 1.0), (0.0, 1.0), (0.0, 0.0), (1.0, 0.0)]
```



> 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.

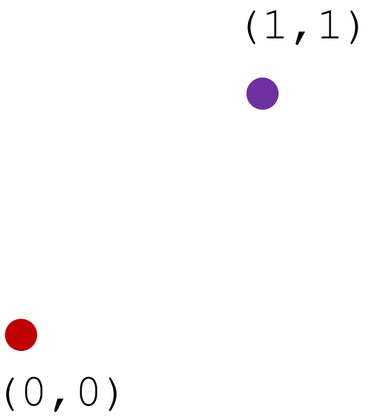
```
a = LineString([(0, 0), (1, 1), (1,2), (2,2)])
b = LineString([(0, 0), (1, 1), (2,1), (2,2)])
x = a.intersection(b)
x
<shapely.geometry.collection.GeometryCollection object at 0x...>

x.wkt
'GEOMETRYCOLLECTION (POINT (2 2), LINESTRING (0 0, 1 1))'

list(x)
[<shapely.geometry.point.Point at 0x107fdc470>,
 <shapely.geometry.linestring.LineString at 0x107fdc588>]
```



> MultiPoint

A MultiPoint object represents a collection of points and consists of a list of coordinate-tuples

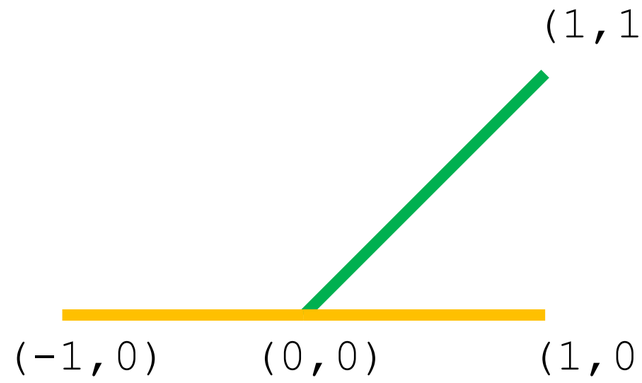
```
from shapely.geometry import MultiPoint
points = MultiPoint([(0.0, 0.0), (1.0, 1.0)])
points.area
0.0
points.length
0.0

points.geoms
<shapely.geometry.base.GeometrySequence at 0x115e37ac8>

list(points.geoms)
[<shapely.geometry.point.Point at 0x115e93048>,
  <shapely.geometry.point.Point at 0x115e93080>]

MultiPoint([Point(0, 0), Point(1, 1)])
```



> MultiLineString

MultiLineString -object represents a collection of lines and consists of a list of line-like sequences

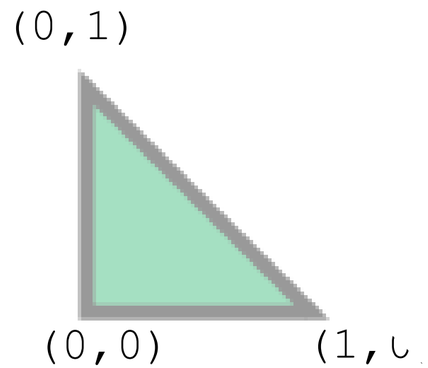
```
from shapely.geometry import MultiLineString
coords = [((0, 0), (1, 1)), ((-1, 0), (1, 0))]
lines = MultiLineString(coords)
lines.area
0.0
lines.length
3.414213562373094 9

len(lines.geoms)
2


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



> MultiPolygon

It takes a sequence of exterior ring and hole list tuples: [((a1, …, aM), [(b1, …, bN), …]), …]. Otherwise it accepts an unordered sequence of Polygon instances.

```
from shapely.geometry import
MultiPolygon
p1=Polygon([(0,0),(0,1),(1,0)])
p2=Polygon([(1,1),(2,0),(3,1),(2,2)])

polygons=MultiPolygon([p1,p2])

len(polygons.geoms)
2

len(polygons)
2
```



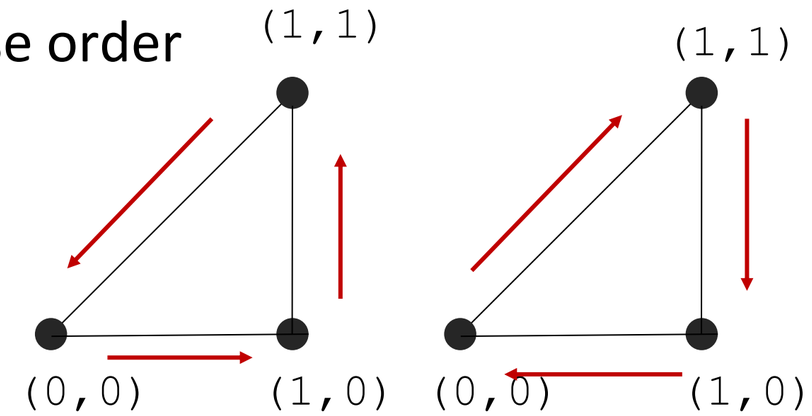
> Predicates

`object.has_z`: True if the feature has not only *x* and *y*, but also *z* coordinates

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

`object.is_ccw`: True if coordinates are in counter-clockwise order

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



`object.is_empty`: True if the feature's *interior* and *boundary* coincide with the empty set.

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

> Predicates

`object.is_ring`: True if the feature is closed.

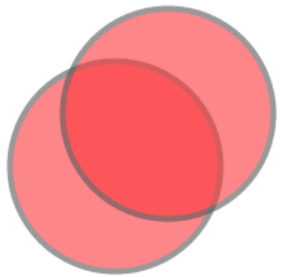
```
LineString([(0, 0), (1, 1), (1, -1)]).is_ring
False
LinearRing([(0, 0), (1, 1), (1, -1)]).is_ring
True
```

`object.is_simple`: True if the feature does not cross itself.

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

`object.is_valid`: True if a feature is “valid”, e.g. a valid *Polygon* may not possess any overlapping exterior or interior rings. A valid MultiPolygon may not collect overlapping polygons

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

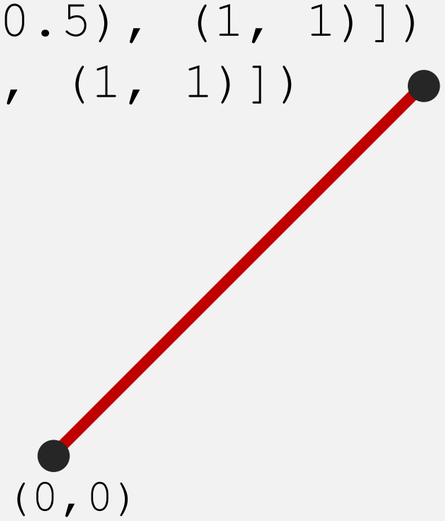


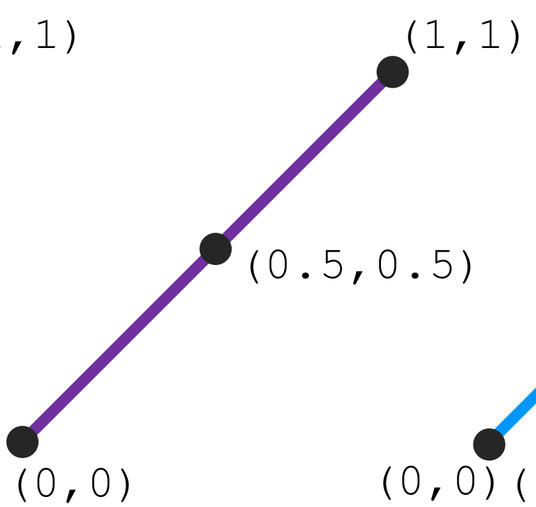
> Predicates

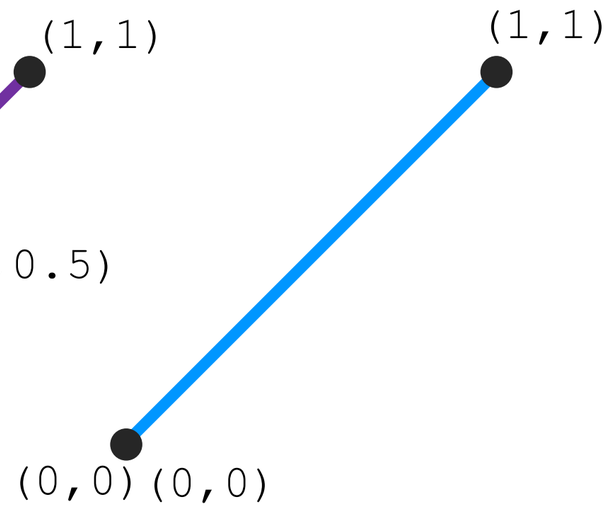
`object.__eq__(other)` : True if the same geometric type, and coordinates match precisely.

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

```
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
```



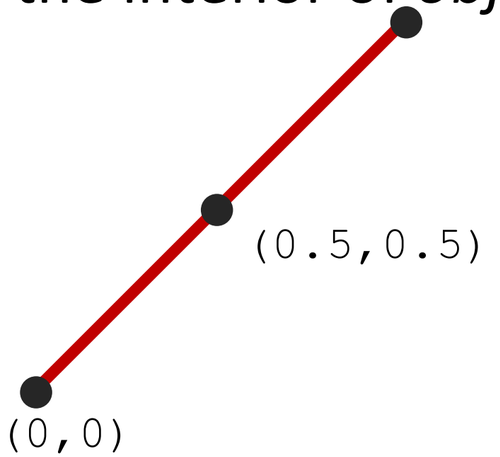




> Predicates

`object.contains(other)` : 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*.

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

Point(0.5, 0.5).within(LineString(coords))
True
```

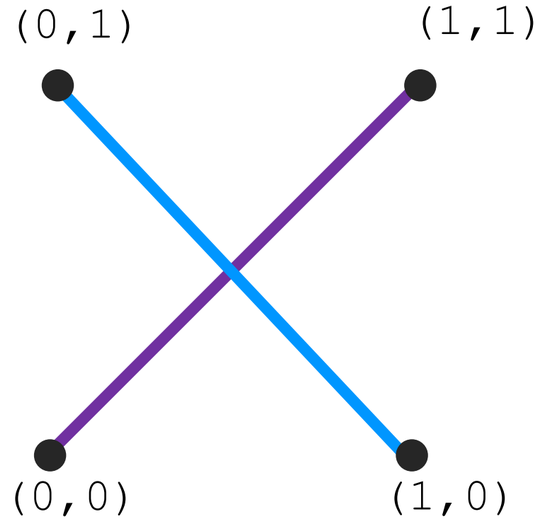


A line's endpoints are part of its *boundary* and are therefore not contained.

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

> Spatial Operations

`object.crosses(other)` : True if the *interior* of the object intersects the *interior* of the other but does not contain it.




```
coords = [(0, 0), (1, 1)]
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)` : 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
```

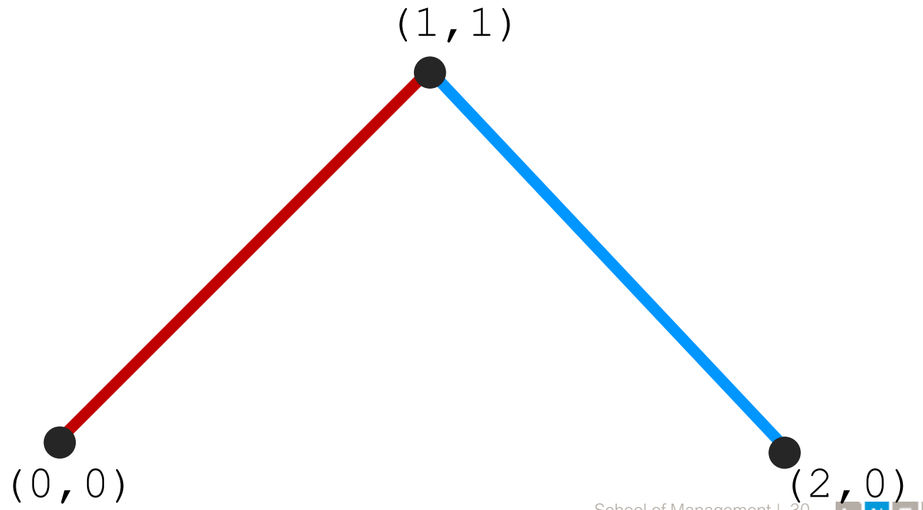
> Spatial Operations

`object.intersects(other)` : True if the *boundary* or *interior* of the object intersect in any way with those of the other.

`object.overlaps(other)` : True if the objects intersect but neither contains the other.

`object.touches(other)` : 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*.

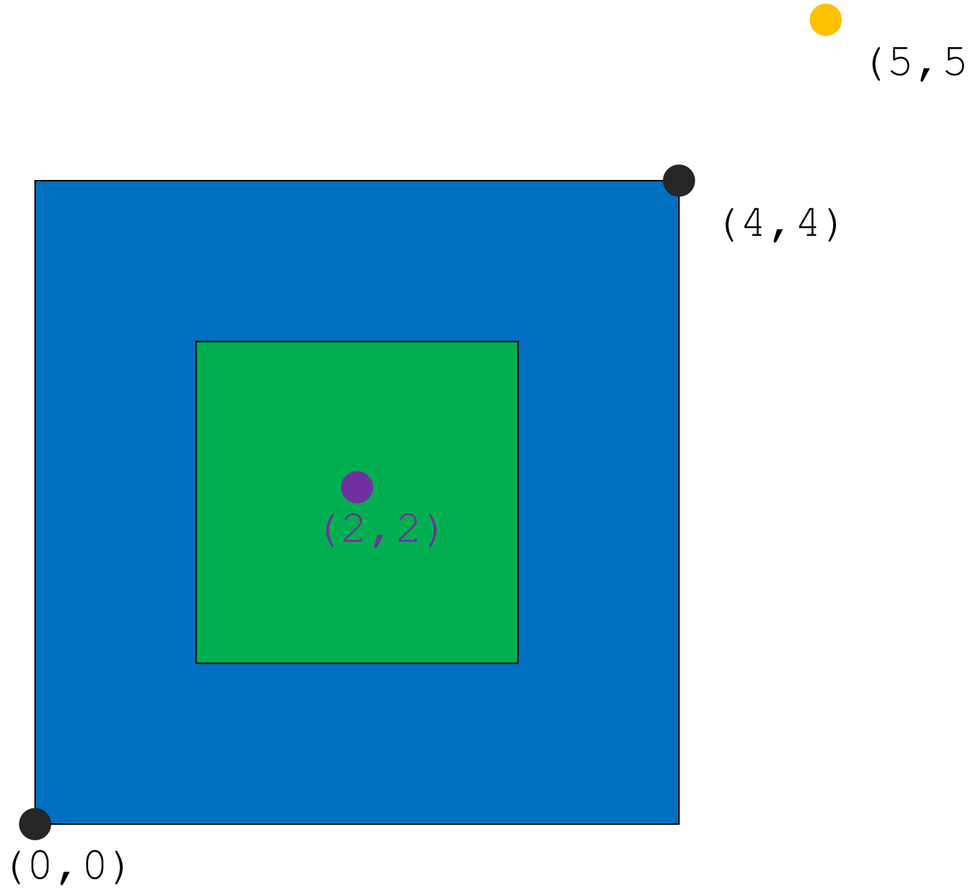
```
a = LineString([(0, 0), (1, 1)])
b = LineString([(1, 1), (2, 0)])
a.touches(b)
True
```



> Spatial Operations

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

```
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(5, 5)

a.within(c)
True

d.within(c)
False

b.within(c)
True
```



> Spatial Operations

`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.

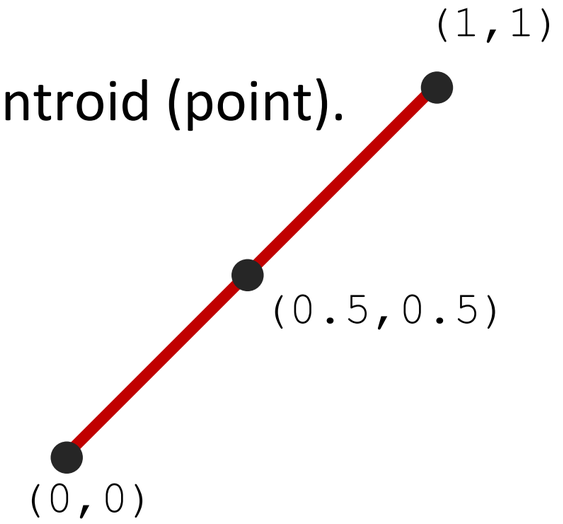
The boundary of a point is an empty (null) collection.

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

lines.boundary.wkt
'MULTIPOINT (-1 0, 0 0, 1 0, 1 1)'

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

> Spatial Operations

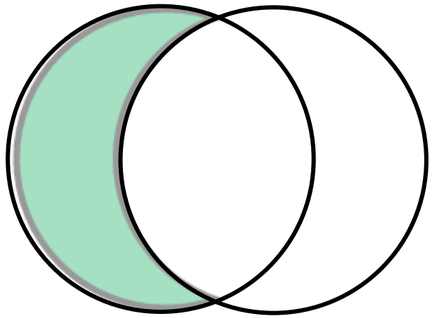
`object.centroid`: Returns a representation of the object's geometric centroid (point).

```
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)'
```



`object.difference(other)`: Returns a representation of the points making up this geometric object that do not make up the *other* object.

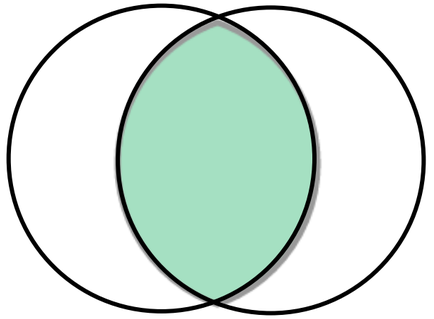
```
a = Point(1, 1).buffer(1.5)
b = Point(2, 1).buffer(1.5)
a.difference(b)
<shapely.geometry.polygon.Polygon object at 0x...>
```



> Construction Operations

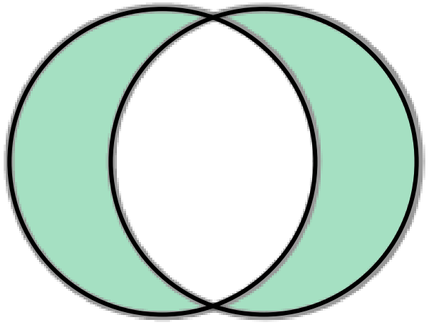
`object.intersection(`*other*`)` : Returns a representation of the intersection of this object with the *other* geometric object.

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



`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.

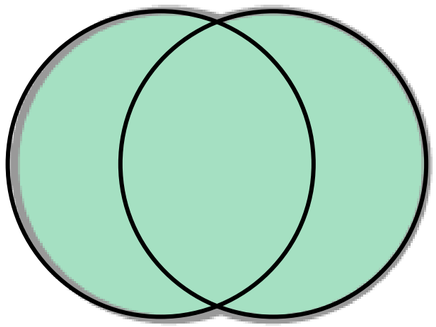
```
a = Point(1, 1).buffer(1.5)
b = Point(2, 1).buffer(1.5)
a.symmetric_difference(b)
<shapely.geometry.multipolygon.MultiPolygon object at ...>
```



> Construction Operations

`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. E.g. the union of polygons will be a polygon or a multi-polygon if they intersect or not.

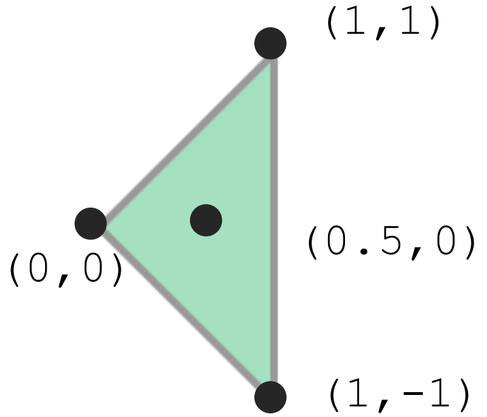
```
a = Point(1, 1).buffer(1.5)
b = Point(2, 1).buffer(1.5)
a.union(b)
<shapely.geometry.polygon.Polygon object at 0x...>
```



> Construction Operations

`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.

```
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), (0.5,0), (1, 1), (1, -1)]).convex_hull
<shapely.geometry.polygon.Polygon object at 0x...>
```



`object.envelope`: Returns a representation of the point or smallest rectangular polygon (with sides parallel to the coordinate axes) that contains the object.

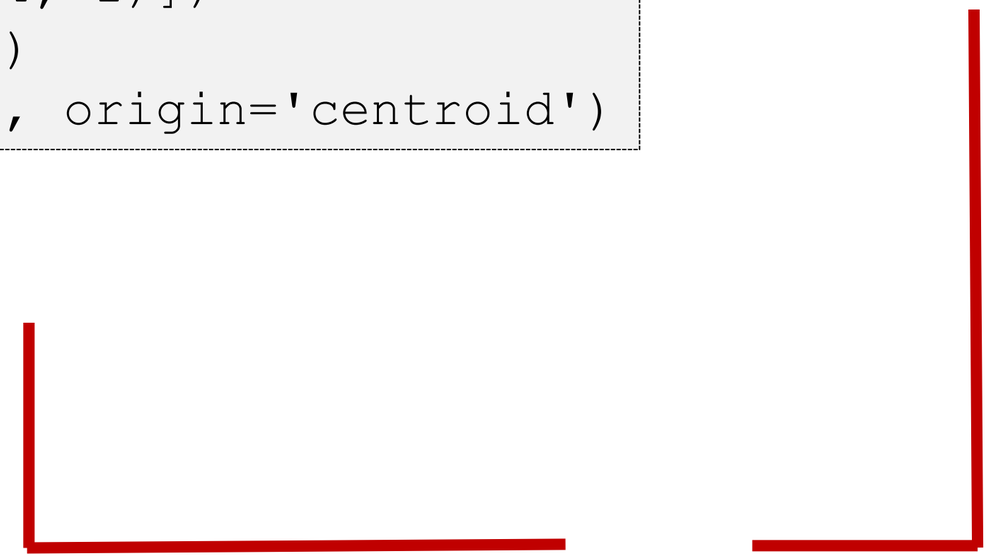
```
MultiPoint([(0, 0), (1, 1)]).envelope
<shapely.geometry.polygon.Polygon object at 0x...>
```

> Transformations

`shapely.affinity.rotate(geom, angle, origin='center', use_radians=False)`: Returns a rotated geometry on a 2D plane.

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).

```
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')
```

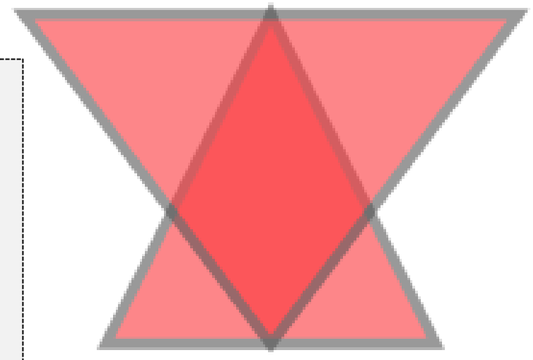


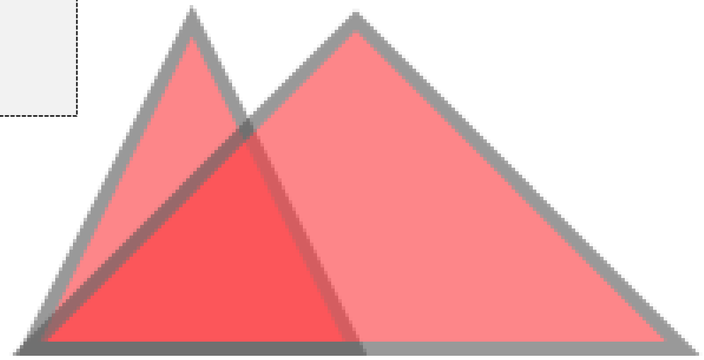
> Transformations

`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.

```
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)]
```





hes·
so
you.

Thank you for your attention.

School of Management
Route de la Plaine 2
3960 Sierre