Shapely and geometric objects

In this lesson, you will learn how to create and manipulate geometries in Python using the Shapely Python Package (https://shapely.readthedocs.io/en/stable/manual.html).

Sources:

These materials are partly based on Shapely-documentation (https://shapely.readthedocs.io/en/stable/manual.html) and Westra E. (2013), Chapter 3 (https://www.packtpub.com/application-development/python-geospatial-development-second-edition).

Spatial data model

Spatial data model

Fundamental geometric objects that can be used in Python with Shapely (https://shapely.readthedocs.io/en/stable/manual.html).

The most fundamental geometric objects are `Points`, `Lines` and `Polygons` which are the basic ingredients when working with spatial data in vector format. Python has a specific module called Shapely (https://shapely.readthedocs.io/en/stable/manual.html) for doing various geometric operations. Basic knowledge of using Shapely is fundamental for understanding how geometries are stored and handled in GeoPandas.

Geometric objects consist of coordinate tuples where:

- `Point` -object represents a single point in space. Points can be either two-dimensional (x, y) or three dimensional (x, y, z).
- `LineString` -object (i.e. a line) represents a sequence of points joined together to form a line. Hence, a line consist of a list of at least two coordinate tuples
- `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.

It is also possible to have a collection of geometric objects (e.g. Polygons with multiple parts):

- `MultiPoint` -object represents a collection of points and consists of a list of coordinate-tuples
- `MultiLineString` -object represents a collection of lines and consists of a list of line-like sequences
- `MultiPolygon` -object represents a collection of polygons that consists of a list of polygon-like sequences that construct from exterior ring and (possible) hole list tuples

Useful attributes and methods in Shapely include:

- Creating lines and polygons based on a collection of point objects.
- Calculating areas/length/bounds etc. of input geometries
- Conducting geometric operations based on the input geometries such as `union`, `difference`, `distance` etc.

- Conducting spatial queries between geometries such as `intersects`, `touches`, `crosses`, `within` etc.

Tuple [Tuple](https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences) is a Python data structure that consists of a number of values separated by commas. Coordinate pairs are often represented as a tuple. For example: ``` (60.192059, 24.945831) ``` Tuples belong to [sequence data types](https://docs.python.org/3/library/stdtypes.html#typesseq) in Python. Other sequence data types are lists and ranges. Tuples have many similarities with lists and ranges, but they are often used for different purposes. The main difference between tuples and lists is that tuples are [immutable](https://docs.python.org/3/glossary.html#term-immutable), which means that the contents of a tuple cannot be altered (while lists are mutable; you can, for example, add and remove values from lists).

Point

Creating point is easy, you pass x and y coordinates into `Point()` -object (+ possibly also z -coordinate):

In [15]:
```
# Import necessary geometric objects from shapely module
from shapely.geometry import Point, LineString, Polygon

# Create Point geometric object(s) with coordinates
point1 = Point(2.2, 4.2)
point2 = Point(7.2, -25.1)
point3 = Point(9.26, -2.456)
point3D = Point(9.26, -2.456, 0.57)
```

Let's see what these variables now contain:

In [16]:
```
point1
```

Out[16]:

As we see here, Jupyter notebook is able to display the shape directly on the screen.

We can use the print statement to get information about the actual definition of these objects:

In [4]:
```
print(point1)
print(point3D)
```

```
POINT (2.2 4.2)
POINT Z (9.26 -2.456 0.57)
```

3D-point can be recognized from the capital Z -letter in front of the coordinates.

Let's also check the data type of a point:

In [17]:
```
type(point1)
```

Out[17]:
```
shapely.geometry.point.Point
```

We can see that the type of the point is shapely's Point. The point object is represented in a specific format based on GEOS (https://trac.osgeo.org/geos) C++ library that is one of the standard libraries behind various Geographic Information Systems. It runs under the hood e.g. in QGIS (http://www.qgis.org/en/site/).

Point attributes and functions

Points and other shapely objects have useful built-in attributes and methods (https://shapely.readthedocs.io/en/stable/manual.html#general-attributes-and-methods). Using the available attributes, we can for example extract the coordinate values of a Point and calculate the Euclidian distance between points.

`geom_type` attribute contains information about the geometry type of the Shapely object:

In [18]:
```
point1.geom_type
```

Out[18]:
```
'Point'
```

Extracting the coordinates of a Point can be done in a couple of different ways:

`coords` attribute contains the coordinate information as a `CoordinateSequence` which is another data type related to Shapely.

In [20]:
```
# Get xy coordinate tuple
list(point1.coords)
```

Out[20]:
```
[(2.2, 4.2)]
```

Here we have a coordinate tuple inside a list. Using the attributes `x` and `y` it is possible to get the coordinates directly as plain decimal numbers.

In [23]:
```
# Read x and y coordinates separately
x = point1.x
y = point1.y
z = point3D.z
```

In [24]:
```
print( x, y, z)
```

```
2.2 4.2 0.57
```

It is also possible to calculate the distance between two objects using the distance (https://shapely.readthedocs.io/en/stable/manual.html#object.distance) method. In our example the distance is calculated in a cartesian coordinate system. When working with real GIS data the distance is based on the used coordinate reference system. always check what is the unit of measurement (for example, meters) in the coordinate reference system you are using.

Let's calculate the distance between `point1` and `point2` :

In [25]:
```
# Check input data
print(point1)
print(point2)
```

```
POINT (2.2 4.2)
POINT (7.2 -25.1)
```

In [27]:
```
# Calculate the distance between point1 and point2
dist = point1.distance(point2)
print(dist)

# Print out a nicely formatted info message
print("Distance between the points is {0:.2f} units".format(dist))
```

```
29.723559679150142
Distance between the points is 29.72 units
```

LineString

Creating LineString -objects is fairly similar to creating Shapely Points.

Now instead using a single coordinate-tuple we can construct the line using either a list of shapely Point -objects or pass the points as coordinate-tuples:

In [28]:
```
# Create a LineString from our Point objects
line = LineString([point1, point2, point3])
```

In [29]:
```
# It is also possible to produce the same outcome using coordinate tup
les
line2 = LineString([(2.2, 4.2), (7.2, -25.1), (9.26, -2.456)])
```

In [31]:
```
# Check if lines are identical
line == line2
```

Out[31]:
```
True
```

Let's see how our line looks like:

In [32]:
```
line
```

Out[32]:

In [36]:
```
print(line)
```

```
LINESTRING (2.2 4.2, 7.2 -25.1, 9.26 -2.456)
```

As we can see from above, the `line` -variable constitutes of multiple coordinate-pairs.

Check also the data type:

In [37]:
```
# Check data type of the line object
type(line)
```

Out[37]:
```
shapely.geometry.linestring.LineString
```

In [38]:
```
# Check geometry type of the line object
line.geom_type
```

Out[38]:
```
'LineString'
```

LineString attributes and functions

`LineString` -object has many useful built-in attributes and functionalities. It is for instance possible to extract the coordinates or the length of a LineString (line), calculate the centroid of the line, create points along the line at specific distance, calculate the closest distance from a line to specified Point and simplify the geometry. See full list of functionalities from Shapely documentation (http://toblerity.org/shapely/manual.html). Here, we go through a few of them.

We can extract the coordinates of a LineString similarly as with `Point`

In [39]:
```
# Get xy coordinate tuples
list(line.coords)
```

Out[39]:
```
[(2.2, 4.2), (7.2, -25.1), (9.26, -2.456)]
```

Again, we have a list of coordinate tuples (x,y) inside a list.

If you would need to access all x-coordinates or all y-coordinates of the line, you can do it directly using the `xy` attribute:

In [40]:
```
# Extract x and y coordinates separately
xcoords = list(line.xy[0])
ycoords = list(line.xy[1])
```

In [41]:
```
print(xcoords)
print(ycoords)
```

```
[2.2, 7.2, 9.26]
[4.2, -25.1, -2.456]
```

It is possible to retrieve specific attributes such as lenght of the line and center of the line (centroid) straight from the LineString object itself:

In [42]:
```
# Get the lenght of the line
l_length = line.length
print("Length of our line: {0:.2f} units".format(l_length))
```

```
Length of our line: 52.46 units
```

In [48]:
```
# Get the centroid of the line
line.centroid
#print(line.centroid)
```

Out[48]:

As you can see, the centroid of the line is again a Shapely Point object.

Polygon

Creating a `Polygon` -object continues the same logic of how `Point` and `LineString` were created but Polygon object only accepts a sequence of coordinates as input.

Polygon needs **at least three coordinate-tuples** (three points are reguired to form a surface):

In [49]:
```
# Create a Polygon from the coordinates
poly = Polygon([(2.2, 4.2), (7.2, -25.1), (9.26, -2.456)])
```

We can also use information from the Shapely Point objects created earlier, but we can't use the point objects directly. Instead, we need to get information of the x,y coordinate pairs as a sequence. We can achieve this by using a list comprehension.

In [58]:
```
# Create a Polygon based on information from the Shapely points
poly2 = Polygon([[p.x, p.y] for p in [point1, point2, point3]])
poly3 = Polygon([point1,point2,point3])
```

In order to understand what just happened, let's check what the list comprehension produces:

In [51]:
```
[[p.x, p.y] for p in [point1, point2, point3]]
```

Out[51]:
```
[[2.2, 4.2], [7.2, -25.1], [9.26, -2.456]]
```

This list of lists was passed as input for creating the Polygon.

In [62]:
```
# Check that polygon objects created using two different approaches ar
e identical
poly == poly2 == poly3
```

Out[62]:
```
True
```

Let's see how our Polygon looks like

In [61]:
```
poly2
```

Out[61]:

In [57]:
```
print(poly)
```

```
POLYGON ((2.2 4.2, 7.2 -25.1, 9.26 -2.456, 2.2 4.2))
```

Notice that `Polygon` representation has double parentheses around the coordinates (i.e. `POLYGON ((<values in here>))`). This is because Polygon can also have holes inside of it.

Check also the data type:

In [64]:
```
# Data type
type(poly)
```

Out[64]:
```
shapely.geometry.polygon.Polygon
```

In [65]:
```
# Geometry type
poly.geom_type
```

Out[65]:
```
'Polygon'
```

In [66]:
```
# Check the help for Polygon objects:
#help(Polygon)
```

As the help of Polygon (https://shapely.readthedocs.io/en/stable/manual.html#polygons) -object tells, a Polygon can be constructed using exterior coordinates and interior coordinates (optional) where the interior coordinates creates a hole inside the Polygon:

```
Help on Polygon in module shapely.geometry.polygon object:
     class Polygon(shapely.geometry.base.BaseGeometry)
      |  A two-dimensional figure bounded by a linear ring
      |
      |  A polygon has a non-zero area. It may have one or more negative-s
pace
      |  "holes" which are also bounded by linear rings. If any rings cros
s each
      |  other, the feature is invalid and operations on it may fail.
      |
      |  Attributes
      |  ----------
      |  exterior : LinearRing
      |      The ring which bounds the positive space of the polygon.
      |  interiors : sequence
      |      A sequence of rings which bound all existing holes.
```

Let's see how we can create a `Polygon` with a hole:

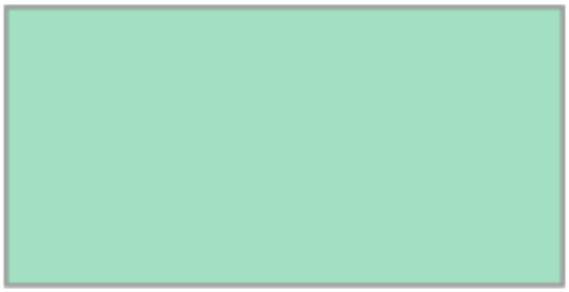
In [67]:
```
# Define the outer border
border = [(-180, 90), (-180, -90), (180, -90), (180, 90)]
```

In [69]:
```
# Outer polygon
world = Polygon(shell=border)
print(world)
```

```
POLYGON ((-180 90, -180 -90, 180 -90, 180 90, -180 90))
```

In [70]:
```
world
```

Out[70]:



In [71]:
```
# Let's create a single big hole where we leave ten units at the bound
aries
# Note: there could be multiple holes, so we need to provide list of c
oordinates for the hole inside a list
hole = [[(-170, 80), (-170, -80), (170, -80), (170, 80)]]
```

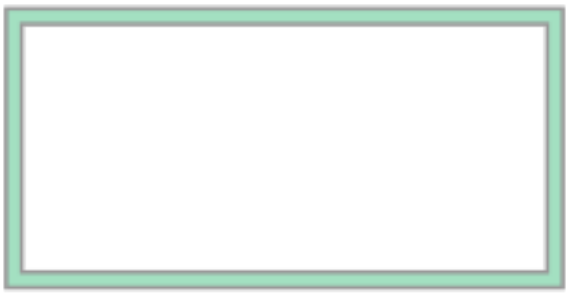
In [72]:
```
# Now we can construct our Polygon with the hole inside
frame = Polygon(shell=border, holes=hole)
print(frame)
```

```
POLYGON ((-180 90, -180 -90, 180 -90, 180 90, -180 90), (-170 80, -17
0 -80, 170 -80, 170 80, -170 80))
```

Let's see what we have now:

In [73]:
```
frame
```

Out[73]:



As we can see the `Polygon` has now two different tuples of coordinates. The first one represents the **outerior** and the second one represents the **hole** inside of the Polygon.

Polygon attributes and functions

We can again access different attributes directly from the `Polygon` object itself that can be really useful for many analyses, such as `area`, `centroid`, `bounding box`, `exterior`, and `exterior-length`. See a full list of methods in the Shapely User Manual (https://shapely.readthedocs.io/en/stable/manual.html#the-shapely-user-manual).

Here, we can see a few of the available attributes and how to access them:

In [74]:
```
# Print the outputs
print("Polygon centroid: ", world.centroid)
print("Polygon Area: ",  world.area)
print("Polygon Bounding Box: ", world.bounds)
print("Polygon Exterior: ", world.exterior)
print("Polygon Exterior Length: ", world.exterior.length)
```

```
Polygon centroid:  POINT (-0 -0)
Polygon Area:  64800.0
Polygon Bounding Box:  (-180.0, -90.0, 180.0, 90.0)
Polygon Exterior:  LINEARRING (-180 90, -180 -90, 180 -90, 180 90, -1
80 90)
Polygon Exterior Length:  1080.0
```

As we can see above, it is again fairly straightforward to access different attributes from the `Polygon` -object. Note that distance metrics will make more sense when we start working with data in a projected coordinate system.

Check your understanding

Plot these shapes using Shapely!

- **Pentagon**, example coords: `(30, 2.01), (31.91, 0.62), (31.18, -1.63), (28.82, -1.63), (28.09, 0.62)`
- **Triangle**
- **Square**
- **Cicrle**

In [75]:
```
# Pentagon - Coordinates borrowed from this thread: https://tex.stackexchange.com/questions/179843/make-a-polygon-with-automatically-labelled-nodes-according-to-their-coordinates
Polygon([(30, 2.01), (31.91, 0.62), (31.18, -1.63), (28.82, -1.63), (28.09, 0.62)])
```

Out[75]:

In [76]:
```
# Triangle
Polygon([(0,0), (2,4), (4,0)])
```

Out[76]:

In [77]:
```
# Square
Polygon([(0,0), (0,4), (4,4), (4,0)])
```

Out[77]:

In [78]:
```
# Circle (using a buffer around a point)
point = Point((0,0))
point.buffer(1)
```

Out[78]:

Geometry collections (optional)

In some occassions it is useful to store multiple geometries (for example, several points or several polygons) in a single feature. A practical example would be a country that is composed of several islands. In such case, all these polygons share the same attributes on the country-level and it might be reasonable to store that country as geometry collection that contains all the polygons. The attribute table would then contain one row of information with country-level attributes, and the geometry related to those attributes would represent several polygon.

In Shapely, collections of points are implemented by using a MultiPoint -object, collections of curves by using a MultiLineString -object, and collections of surfaces by a MultiPolygon -object.

In [79]:
```
# Import constructors for creating geometry collections
from shapely.geometry import MultiPoint, MultiLineString, MultiPolygon
```

Let's start by creating MultiPoint and MultilineString objects:

In [80]:
```
# Create a MultiPoint object of our points 1,2 and 3
multi_point = MultiPoint([point1, point2, point3])

# It is also possible to pass coordinate tuples inside
multi_point2 = MultiPoint([(2.2, 4.2), (7.2, -25.1), (9.26, -2.456)])

# We can also create a MultiLineString with two lines
line1 = LineString([point1, point2])
line2 = LineString([point2, point3])
multi_line = MultiLineString([line1, line2])

# Print object definitions
print(multi_point)
print(multi_line)
```

```
MULTIPOINT (2.2 4.2, 7.2 -25.1, 9.26 -2.456)
MULTILINESTRING ((2.2 4.2, 7.2 -25.1), (7.2 -25.1, 9.26 -2.456))
```

In [81]:
```
multi_point
```

Out[81]:

In [82]:
```
multi_line
```

Out[82]:

MultiPolygons are constructed in a similar manner. Let's create a bounding box for "the world" by combinin two separate polygons that represent the western and eastern hemispheres.

In [83]:
```
# Let's create the exterior of the western part of the world
west_exterior = [(-180, 90), (-180, -90), (0, -90), (0, 90)]

# Let's create a hole --> remember there can be multiple holes, thus w
e need to have a list of hole(s).
# Here we have just one.
west_hole = [[(-170, 80), (-170, -80), (-10, -80), (-10, 80)]]

# Create the Polygon
west_poly = Polygon(shell=west_exterior, holes=west_hole)

# Print object definition
print(west_poly)
```

```
POLYGON ((-180 90, -180 -90, 0 -90, 0 90, -180 90), (-170 80, -170 -8
0, -10 -80, -10 80, -170 80))
```

In [84]:
```
west_poly
```

Out[84]:

Shapely also has a tool for creating a bounding box (https://en.wikipedia.org/wiki/Minimum_bounding_box) based on minimum and maximum x and y coordinates. Instead of using the Polygon constructor, let's use the box (https://shapely.readthedocs.io/en/stable/manual.html#shapely.geometry.box) constructor for creating the polygon:

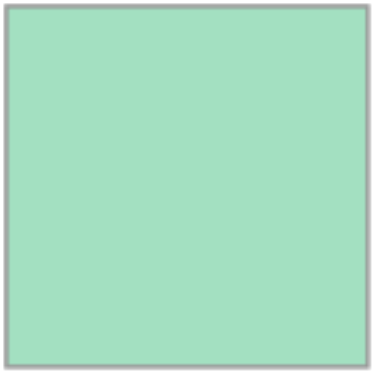
In [85]:
```
from shapely.geometry import box
```

In [86]:
```
# Specify the bbox extent (lower-left corner coordinates and upper-rig
ht corner coordinates)
min_x, min_y = 0, -90
max_x, max_y = 180, 90

# Create the polygon using Shapely
east_poly = box(minx=min_x, miny=min_y, maxx=max_x, maxy=max_y)

# Print object definition
print(east_poly)
```

```
POLYGON ((180 -90, 180 90, 0 90, 0 -90, 180 -90))
```

In [87]:
```
east_poly
```

Out[87]:



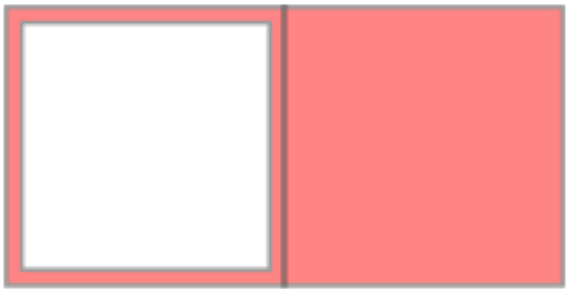
Finally, we can combine the two polygons into a MultiPolygon:

In [88]:
```
# Let's create our MultiPolygon. We can pass multiple Polygon -objects
into our MultiPolygon as a list
multi_poly = MultiPolygon([west_poly, east_poly])

# Print object definition
print(multi_poly)
```

```
MULTIPOLYGON (((-180 90, -180 -90, 0 -90, 0 90, -180 90), (-170 80, -
170 -80, -10 -80, -10 80, -170 80)), ((180 -90, 180 90, 0 90, 0 -90,
180 -90)))
```

In [89]:
```
multi_poly
```

Out[89]:



We can see that the outputs are similar to the basic geometric objects that we created previously but now these objects contain multiple features of those points, lines or polygons.

Convex hull and envelope

Convex hull refers to the smalles possible polygon that contains all objects in a collection. Alongside with the minimum bounding box, convex hull is a useful shape when aiming to describe the extent of your data.

Let's create a convex hull around our multi_point object:

In [90]:
```
# Check input geometry
multi_point
```

Out[90]:

In [91]:
```
# Convex Hull (smallest polygon around the geometry collection)
multi_point.convex_hull
```

Out[91]:

In [92]:
```
# Envelope (smalles rectangular polygon around the geometry collectio
n):
multi_point.envelope
```

Out[92]:

Other useful attributes

lenght of the geometry collection:

In [93]:
```
print("Number of objects in our MultiLine:", len(multi_line))
print("Number of objects in our MultiPolygon:", len(multi_poly))
```

```
Number of objects in our MultiLine: 2
Number of objects in our MultiPolygon: 2
```

Area:

In [94]:
```
# Print outputs:
print("Area of our MultiPolygon:", multi_poly.area)
print("Area of our Western Hemisphere polygon:", multi_poly[0].area)
```

```
Area of our MultiPolygon: 39200.0
Area of our Western Hemisphere polygon: 6800.0
```

From the above we can see that MultiPolygons have exactly the same attributes available as single geometric objects but now the information such as area calculates the area of **ALL** of the individual -objects combined. We can also access individual objects inside the geometry collections using indices.

Finally, we can check if we have a "valid" MultiPolygon. MultiPolygon is thought as valid if the individual polygons does notintersect with each other. Here, because the polygons have a common 0-meridian, we should NOT have a valid polygon. We can check the validity of an object from the **is_valid** -attribute that tells if the polygons or lines intersect with each other. This can be really useful information when trying to find topological errors from your data:

In [95]:
```
print("Is polygon valid?: ", multi_poly.is_valid)
```

```
Is polygon valid?:  False
```