# PYQGIS BASICS

*Master in Environmental Management of Mountains Areas*

*ADVANCED GEOMATICS*

*Andrea Antonello - Free University of Bolzano*

*March - June 2024*

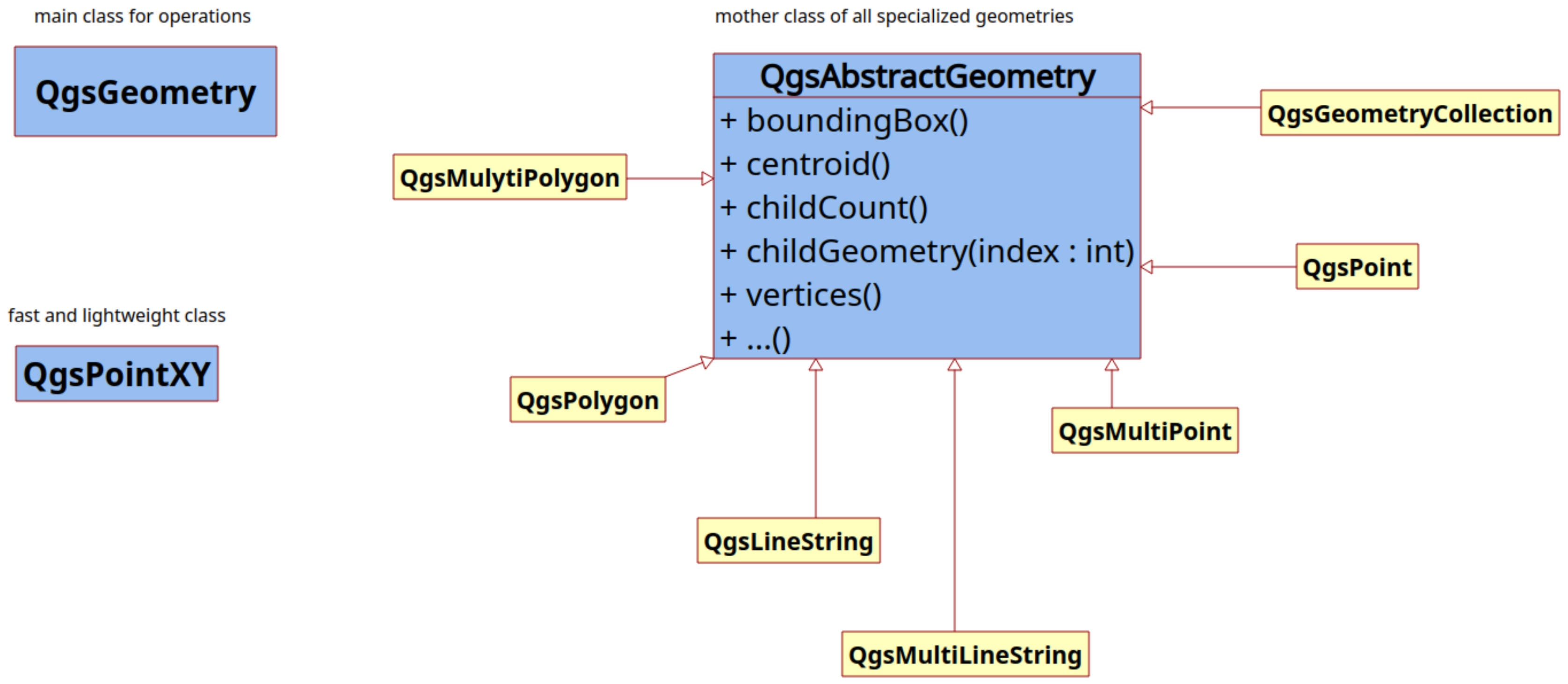
## PYQGIS

## GEOMETRIES

To handle geometries QGIS supplies the following main classes:

- QgsPoint, QgsMultiPoint
- QgsLineString, QgsMultiLineString
- QgsPolygon, QgsMultiPolygon
- QgsGeometry
- QGsPointXY

# CLASS DIAGRAM



## PYQGIS ISSUES: VERBOSITY

Example: style a point layer

```python
properties = {
    'name': 'square',
    'size': 8,
    'color': '255,0,0,128',
    'outline_color': 'black',
    'outline_width': 1,
    'angle': 45
}
symbol = QgsMarkerSymbol.createSimple(properties) 4
layer.renderer().setSymbol(symbol)
```

## Example: put labels

```python
settings = QgsPalLayerSettings()
format = QgsTextFormat()
format.setFont(QFont('Arial', 8))
format.setColor(QColor('red'))
buffer = QgsTextBufferSettings()
buffer.setEnabled(True)
buffer.setSize(0.50)
buffer.setColor(QColor('black'))
format.setBuffer(buffer)
settings.setFormat(format)
settings.fieldName = "NAME"
settings.placement = QgsPalLayerSettings.OverPoint
settings.xOffset = 0.0
settings.yOffset = -8.5
labels = QgsVectorLayerSimpleLabeling(settings)
layer.setLabelsEnabled(True)
layer.setLabeling(labels)
```

## PYQGIS ISSUES: CRASHABILITY

Since pyQGIS is bound to its lowlevel API, memory management can be an issue. If you reuse the same variable name for different objects, most of the times you will experience a QGIS crash.

```python
p = QgsPoint(12, 46)
pg = QgsGeometry(p)

coords = [[31,11], [10,30], [20,40], [40,40]]
l = QgsLineString([ QgsPoint(p[0], p[1]) for p in coords])
lg = QgsGeometry(l)

print(pg)
print(lg)

'''assume somewhere else in the code you just need to recreate
the QgsGeometry for whatever reason, using the previous p object'''

pg = QgsGeometry(p)
print(pg)
```

## PYQGIS SCRIPTING EXTENSIONS

To make the life of the scripter easier, there are some extensions that can be used to make the code more readable and less error prone.

They have been developed to make pyQGIS more scripting like and try to save users from potential crash situations.

The above marker and label example can be written as:

```python
from pyqgis_scripting_ext.core import *

pointStyle = HMarker("square", 8, 45) \
           + HFill("0,255,0,128") + HStroke("black", 1) \
           + HLabel("NAME", yoffset = -5) + HHalo("white", 1)
layer.set_style(pointStyle)
```

## INSTALLING THE EXTENSIONS

The extensions can be installed from the QGIS console directly. Create a new editor and paste the following code:

```python
import pip
pip.main(['install', 'pyqgis-scripting-ext'])
```

This will import the module pip and through it install the pyqgis-scripting-ext module.

This ensures that the extensions are installed in the python environment that QGIS uses.

## CREATE GEOMETRIES

## CREATE A POINT

Geometries in general can be built either from a WKT representation or using directly the list of coordinates:

```
point = HPoint(30.0, 10.0) 1
print(point.asWkt()) 2
```

1. a point is made of a lon and lat value (order is important).
2. to print a geometry in its well known text (WKT) representation, use the **asWkt()** method.

## CREATE A LINESTRING

More complex geometries have a **fromCoords** method to create them from a list of coordinates (pairs of floating point values):

```
coords = [[31,11], [10,30], [20,40], [40,40]] (1)
line = HLineString.fromCoords(coords)
print(line.asWkt())
```

Using the API directly:

```
coords = [[31,11], [10,30], [20,40], [40,40]]
points = []
for coord in coords:
    points.append(QgsPoint(coord[0], coord[1]))

line = QgsLineString(points)
print(line.asWkt())
```

## CREATE A POLYGON

```
coords = [[32,12], [10,20], [20,39], [40,39], [32,12]] (1)
polygon = HPolygon.fromCoords(coords)
```

(1) a polygon is created using a closed linestring. This will be the exterior ring of the polygon.

If the polygon contains holes, they can be passed as **rings**, i.e. closed linestrings:

```
exteriorPoints = [[35,10],[10,20],[15,40],[45,45],[35,10]]
holePoints = [[20,30],[35,35],[30,20],[20,30]]

polygonWithHole = HPolygon.fromCoords(exteriorPoints) (1)

holeRing = HLineString.fromCoords(holePoints) (2)
polygonWithHole.add_interior_ring(holeRing) (3)
print(polygonWithHole.asWkt())
```

1. Create the polygon using the exterior ring.
2. Create the hole ring.
3. Add the hole ring to the polygon.

## GET THE BOUND OF A POLYGON

```
print(polygonWithHole.exterior_ring().asWkt()) 1
print("hole count:", polygonWithHole.interior_rings_count()) 2
print(polygonWithHole.interior_ring(0).asWkt()) 3
```

1. the external bound of a polygon can be retrieved as an HLineString through the **exteriorRing** attribute.
2. since the interior rings can be several, their count can be obtained…
3. and through the index of the ring, the line of the hole.

## VISUALIZE GEOMETRIES

The **asWkt** function of the geometry object helps a lot to "see" the content of a geometry:

```
POLYGON ((35 10, 10 20, 15 40, 45 45, 35 10),
         (20 30, 35 35, 30 20, 20 30))
```

Is this a polygon with a hole or a multi-polygon? Can you tell?

What do all the above geometries look like?

When it comes to geometric data, visual tools come in handy.

## EXPLOITING THE MAP CANVAS

To have a visual feedback, but without having to create real data (which would be vector layers), we can make use of the canvas.

**Create a map canvas**

QGIS allows to create additional map canvas that behave the same way as the main one (zoom, pan, tools, crs). To create and show a new canvas:

```
canvas = HMapCanvas.new()
canvas.show()
```

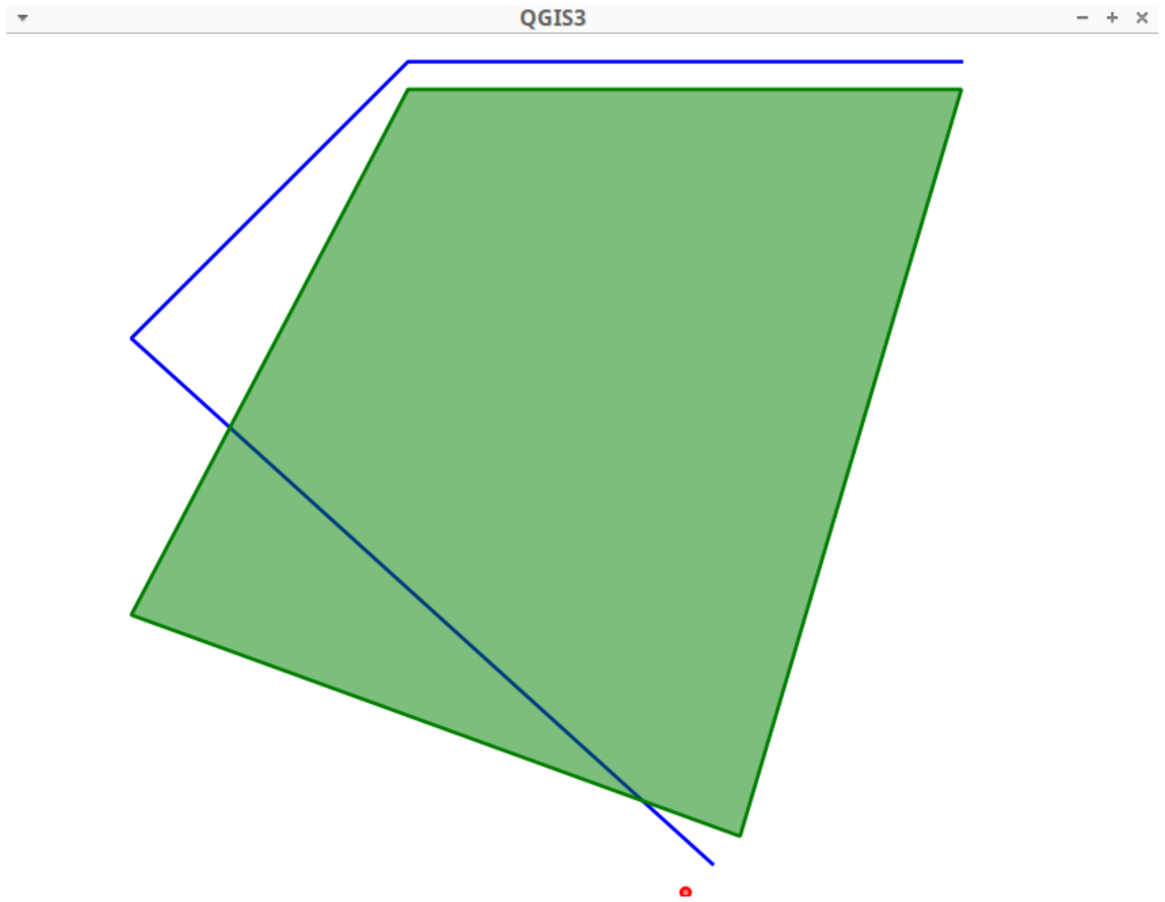
On teh canvas you can draw geometries using a simple style:

```
canvas = HMapCanvas.new()
canvas.add_geometry(point, "red", 2) ❶
canvas.add_geometry(line, "blue", 2)
canvas.add_geometry(polygon, "green", 2)

canvas.set_extent([0, 0, 1000, 1000]) ❷
canvas.show()
```

❶ geometries can be added specifying a color and a width.

❷ the extent of the canvas can be set using a list of [minx, miny, maxx, maxy]

The result is something like:



## CREATING MULTI-GEOMETRIES

Multi-geometries are basically created by adding the single geometries to the multi-geometry:

Multipoint:

```
coords = [[10,40],[40,30],[20,20],[30,10]]
multiPoints = HMultiPoint.fromCoords(coords)
```

MultiLine:

```
coords1 = [[10,10],[20,20],[10,40]]
coords2 = [[40,40],[30,30],[40,20],[30,10]]
multiLine = HMultiLineString.fromCoords([coords1, coords2])
```

MultiPolygon:

```
coords1 = [[30,20], [10,40], [45,40], [30,20]]
coords2 = [[15,5], [40,10], [10,20], [5,10], [15,5]]
multiPolygon = HMultiPolygon.fromCoords([coords1, coords2])
```

or, if you need to add interior rings, first build single polygons:

```
polygon1 = HPolygon.fromCoords(coords1)
polygon1.add_interior_ring(...)
polygon2 = HPolygon.fromCoords(coords2)
multiPolygon = HMultiPolygon([polygon1, polygon2])
```

## SUBGEOMETRIES AND COORDINATES

```
subGeometries = multiPolygon.geometries() (1)
for i in range(len(subGeometries)):
    child = subGeometries[i]
    print(f"polygon at position {i} = {child.asWkt()}")
```

1. every geometry has a method to get a list of its geometries (can be just itself)

## COORDINATES

To access the coordinates of a geometry, use the coordinates method. It returns a list of x/y pairs:

```python
for i, coordinate in enumerate(polygon.coordinates()):
    print(f"coord {i}) x={coordinate[0]}, y={coordinate[1]}")

for i, coordinate in enumerate(line.coordinates()):
    print(f"coord {i}) x={coordinate[0]}, y={coordinate[1]}")
```

## INTERMEZZO: HOW DO I CHECK THE TYPE OF A VARIABLE

Sometimes it is necessary to know exactly of what type a variable is.

This can be printed out just using the **type** function:

```python
firstGeom = multiPolygon.geometries()[0]
print(f"Geometry type: {type(firstGeom)}")
```

If instead a check is necessary as part of the script logic, **isinstance*** can be used to check against a class:

```python
if isinstance(firstGeom, HPolygon):
    print(f"It indeed is a Polygon!")
```

## CREATE GEOMETRIES FROM WKT

It is possible to convert a WKT representation of a geometry to an actual geometry object using the **fromWKT** method of the Geometry class:

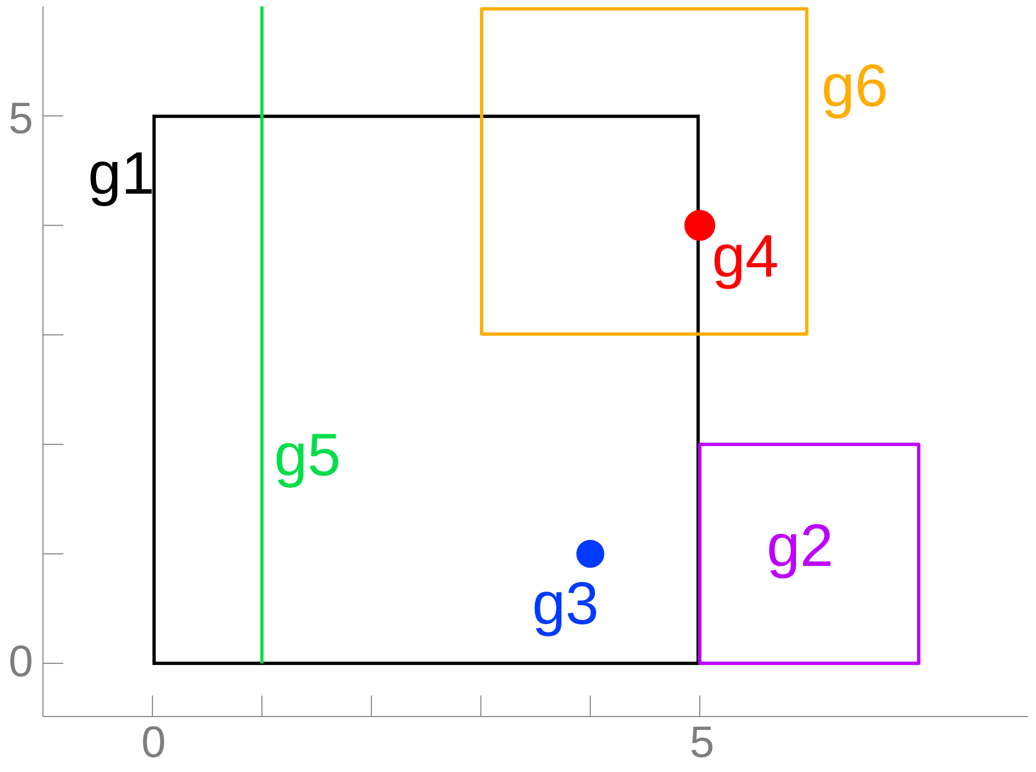
```
wkt = "POINT (156 404)"
pointGeom = HGeometry.fromWkt(wkt)
```

This comes in handy with complex geometries:

```
wkt = """
MULTIPOLYGON (((130 510, 140 450, 200 480, 210 570, 150 630, 130 560, 130 510)),
  ((430 770, 370 820, 210 860, 20 760, 35 631, 100 370, 108 363, 154 284, 230 380,
      140 400, 150 440, 130 450, 104 585, 410 670, 440 590, 450 590, 430 770)))
"""
polygonGeom = HGeometry.fromWkt(wkt)
```

## CREATE A TEST SET

To better explain the various functions and predicates we will start by creating a set of geometries on which to apply the operations. You are now able to create the following points, line and polygons:

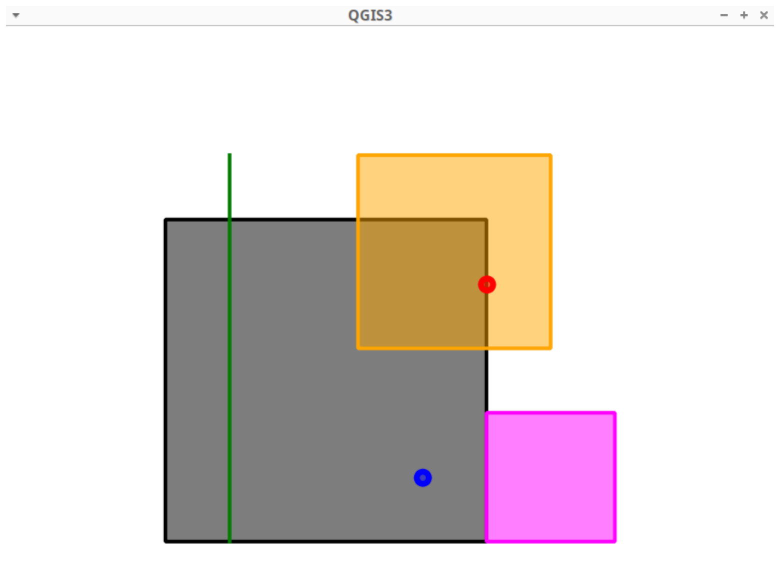


Let’s create the geometries that make up the test set:

```
g1 = HPolygon.fromCoords([[0, 0], [0, 5], [5, 5], [5, 0], [0, 0]])
g2 = HPolygon.fromCoords([[5, 0], [5, 2], [7, 2], [7, 0], [5, 0]])
g3 = HPoint(4, 1)
g4 = HPoint(5, 4)
g5 = HLineString.fromCoords([[1, 0], [1, 6]])
g6 = HPolygon.fromCoords([[3, 3], [3, 6], [6, 6], [6, 3], [3, 3]])
```

## VIEW IT IN A DEDICATED MAPCANVAS

```
canvas = HMapCanvas.new()
canvas.add_geometry(g1, 'black', 3)
canvas.add_geometry(g2, 'magenta', 3)
canvas.add_geometry(g6, 'orange', 3)
canvas.add_geometry(g5, 'green', 3)
canvas.add_geometry(g3, 'blue', 10)
canvas.add_geometry(g4, 'red', 10)
canvas.set_extent([-1, -1, 8, 8])
canvas.show()
```



## THE ENVELOPE OF A GEOMETRY

The envelope of a geometry is given by the **bbox** function.

```
print("polygon bbox:", g1.bbox())
```

It is in the form of a list of four values: [minx, miny, maxx, maxy]

## LENGTH, AREA AND DISTANCE

```
print("polygon length:", g1.length()) ❶
print("polygon area:", g1.area())

print("line length:", g5.length())
print("line area:", g5.area()) ❷

print("point length:", g3.length())
print("point area:", g3.area()) ❸

print("distance between line and point:", g5.distance(g4)) ❹
```

❶ for polygons the **length** is the perimeter.

❷ for lines the area=0,

❸ for points both area and length are 0.

❹ **distance** returns the distance of the two nearest points on the geometries (planar).

## PREDICATES

## INTERSECTS

Let’s see which geometries intersect with g1 and print the result:

```
print(g1.intersects(g2)) # true
print(g1.intersects(g3)) # true
print(g1.intersects(g4)) # true
print(g1.intersects(g5)) # true
print(g1.intersects(g6)) # true
```

Note that geometries that touch (like g1 and g2) also intersect.

## TOUCHES

Let’s test which geometries touch g1 and print the result:

```
print(g1.touches(g2)) # true
print(g1.touches(g3)) # false
print(g1.touches(g4)) # true
print(g1.touches(g5)) # false
print(g1.touches(g6)) # false
```

Definition: the geometries have at least one point in common, but their interiors do not intersect.

## CONTAINS

Let’s test which geometries are contained by g1 and print the result:

```
print(g1.contains(g2)) # false
print(g1.contains(g3)) # true
print(g1.contains(g4)) # false
print(g1.contains(g5)) # false
print(g1.contains(g6)) # false
```

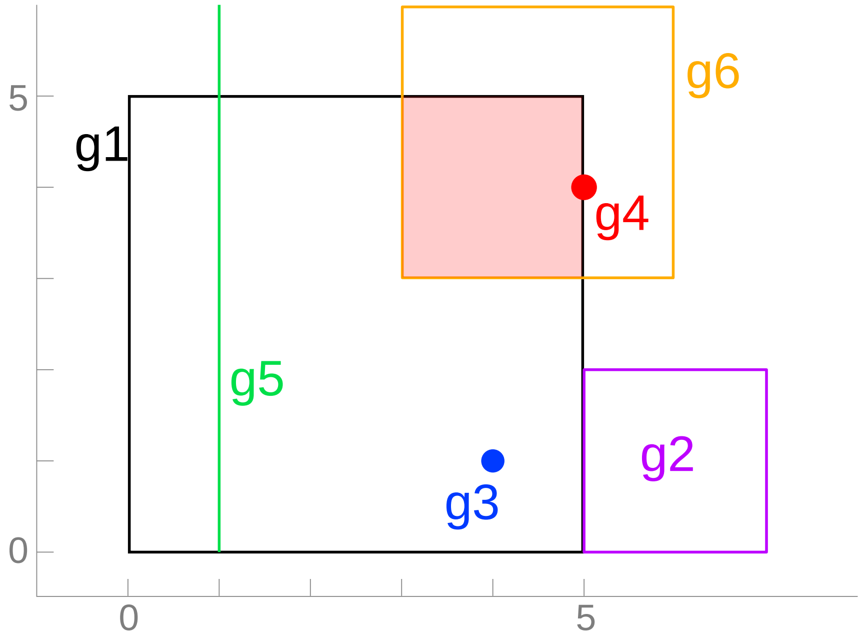
Mind that a point on the border is not contained, so only g3 is contained.

## FUNCTIONS

## INTERSECTION

The intersection of polygons returns a polygon

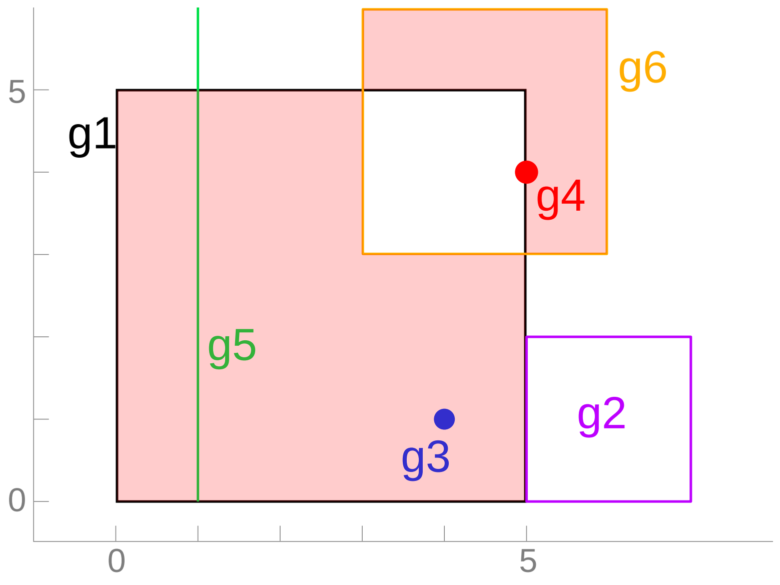
```
print(g1.intersection(g6))
print(g1.intersection(g2)) # touching polygons -> line
print(g1.intersection(g3)) # polygon and point -> point
print(g1.intersection(g5)) # polygon and line -> line
```



## SYMDIFFERENCE

What is the resulting geometry of the symdifference (portions not shared) of different geometry types?

```
# intersecting polygons -> multipolygon
print(g1.symdifference(g6))
# touching polygons -> polygons union
print(g1.symdifference(g2))
# polygon with a contained point -> original polygon
print(g1.symdifference(g3))
# polygon with a line -> hybrid collection (line + polygon)
print(g1.symdifference(g5))
```



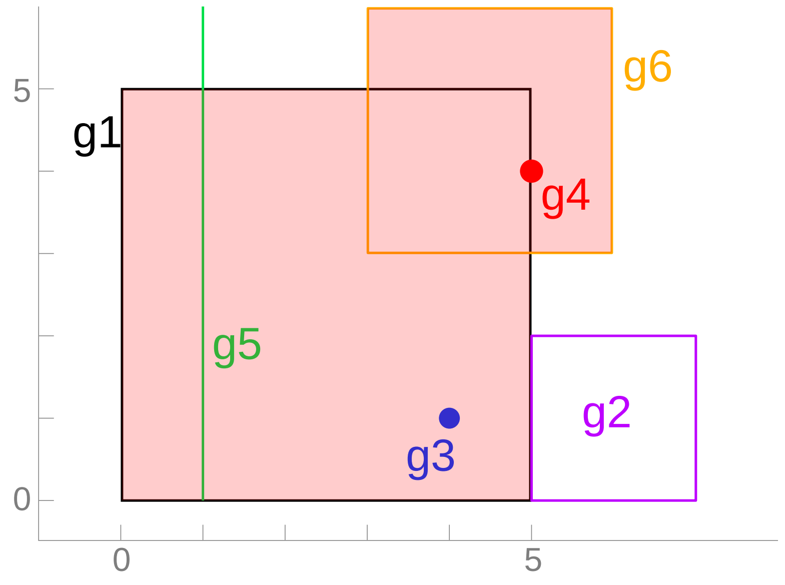
## VISUAL SYMDIFFERENCE

```python
canvas = HMapCanvas.new()
canvas.add_geometry(g1.exterior_ring(), 'black', 3)
canvas.add_geometry(g6.exterior_ring(), 'magenta', 3)
canvas.add_geometry(g1.symdifference(g6), 'orange', 3)
canvas.set_extent([-1, -1, 8, 8])
canvas.show()
```

## COMBINE

What is the resulting geometry of the union of different geometry types?

```
# intersecting polygons -> polygon
print(g1.union(g6))
# same for the union of touching polygons
print(g1.union(g2))
# polygon with contained point -> original polygon
print(g1.union(g3))
# polygon and line -> hybrid collection (line + polygon)
print(g1.union(g5))
```



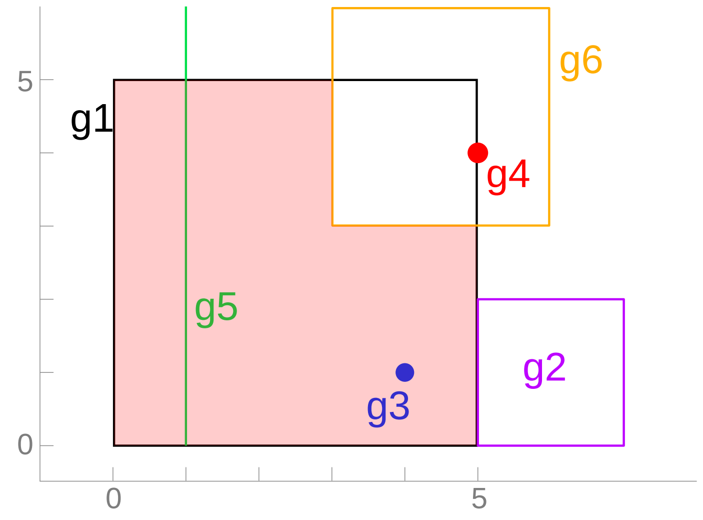
## VISUAL COMBINE

```
canvas = HMapCanvas.new()
canvas.add_geometry(g1.exterior_ring(), 'black', 3)
canvas.add_geometry(g6.exterior_ring(), 'magenta', 3)
canvas.add_geometry(g1.union(g6), 'orange', 3)
canvas.set_extent([-1, -1, 8, 8])
canvas.show()
```

## DIFFERENCE

The difference of geometries obviously depends on the calling object:

```
# this returns g1 minus the overlapping part of g6
print(g1.difference(g6))
# while this returns g6 minus the overlapping part of g1
print(g6.difference(g1))
# in the case of difference with lines -> original polygon
# + additional points in the intersections
print(g1.difference(g5))
# the difference of polygon and point -> original polygon
print(g1.difference(g3))
```



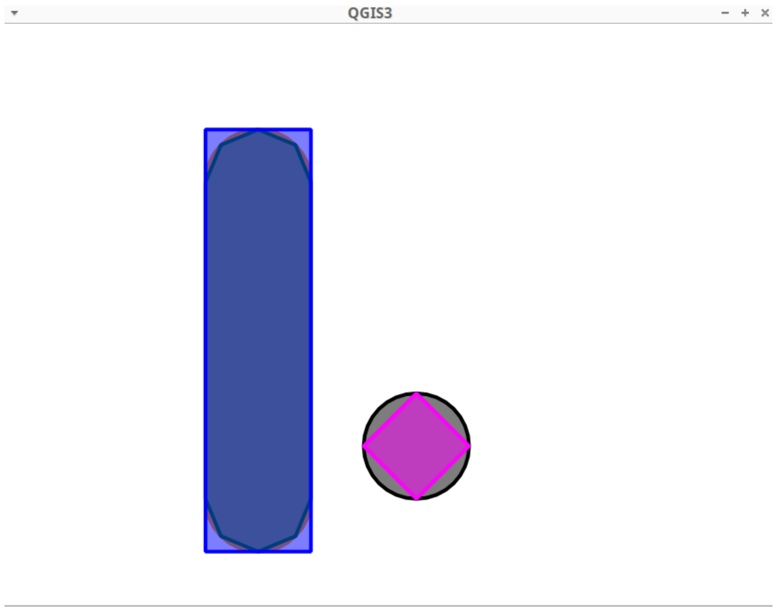
## VISUAL DIFFERENCE

```
canvas = HMapCanvas.new()
canvas.add_geometry(g1.exterior_ring(), 'black', 3)
canvas.add_geometry(g6.exterior_ring(), 'magenta', 3)
canvas.add_geometry(g6.difference(g1), 'orange', 3)
canvas.set_extent([-1, -1, 8, 8])
canvas.show()
```

# BUFFER

Creating a buffer around a geometry always generates polygonal geometries. The behaviour can be tweaked, depending on the geometry type:

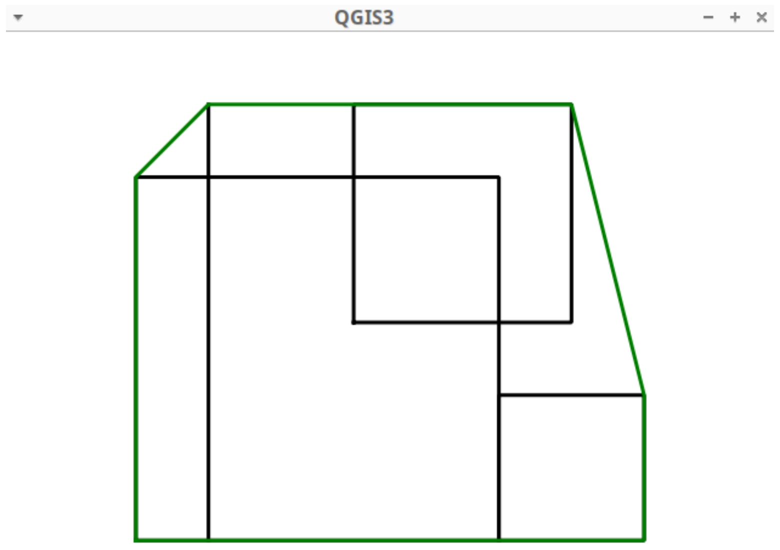
```
b1 = g3.buffer(1.0) # the buffer of a point
b2 = g3.buffer(1.0, 1) # the buffer of a point with few quandrant segments
b3 = g5.buffer(1.0)  # line buffer
b4 = g5.buffer(1.0, 2)  # line buffer with few points
# square end cap style (flat, square, round)
b5 = g5.buffer(1.0, -1, JOINSTYLE_ROUND, ENDCAPSTYLE_SQUARE)
```



# CONVEX HULL

Let’s create a geometry collection containing all the geometries. Then apply the convex hull function:

```
collection = HGeometryCollection([g1, g2, g3, g4, g5, g6])
convexhull = collection.convex_hull()
```



```
canvas = HMapCanvas.new()
canvas.add_geometry(b1, 'black', 3)
canvas.add_geometry(b2, 'magenta', 3)
canvas.add_geometry(b3, 'orange', 3)
canvas.add_geometry(b4, 'green', 3)
canvas.add_geometry(b5, 'blue', 3)
canvas.add_geometry(convexhull.exterior_ring(), 'brown', 5)
canvas.set_extent([-2, -2, 9, 9])
canvas.show()
```

## PROJECTIONS

## REPROJECT GEOMETRIES

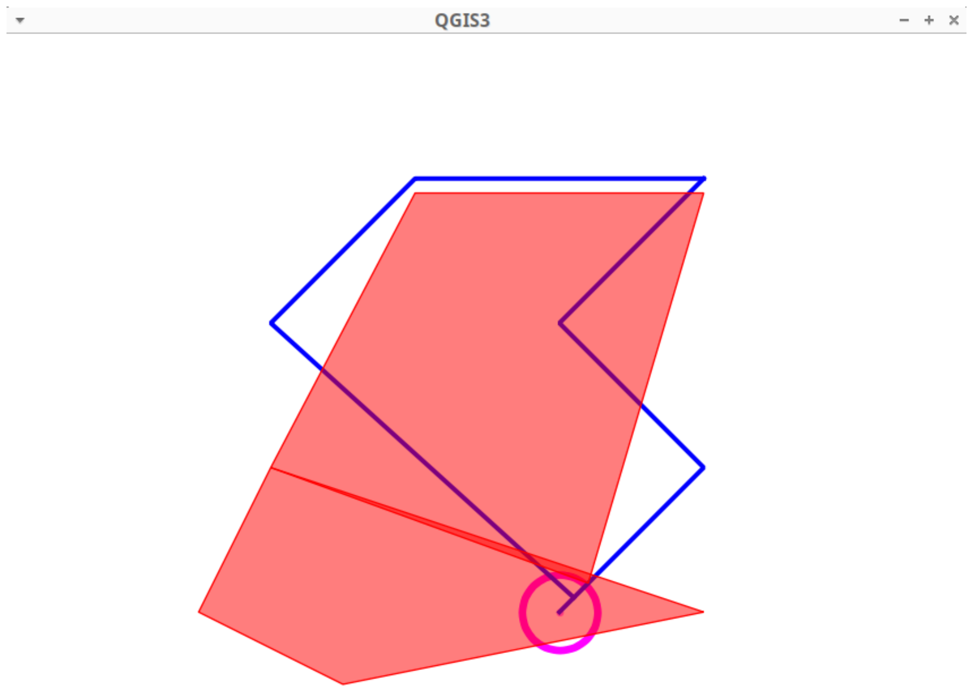
```
crsHelper = HCrs() 1
crsHelper.from_srid(4326)
crsHelper.to_srid(32632)
point4326 = HPoint(11, 46)
point32632 = crsHelper.transform(point4326) 2

print(f"{point4326.asWkt()} -> {point32632.asWkt()}'")

backTo4326 = crsHelper.transform(point32632, inverse = True) 3
print(backTo4326.asWkt())
```

1. create the crs helper object and set start and destination projection
2. transform a geometry
3. it is also possible to use the reverse transformation

## EXERCISES

## 00 - DRAW GEOMETRIES ON A MAP CANVAS

Write a script that reads geometries from the file **02_exe0_geometries.csv** and draws them on a new map canvas.

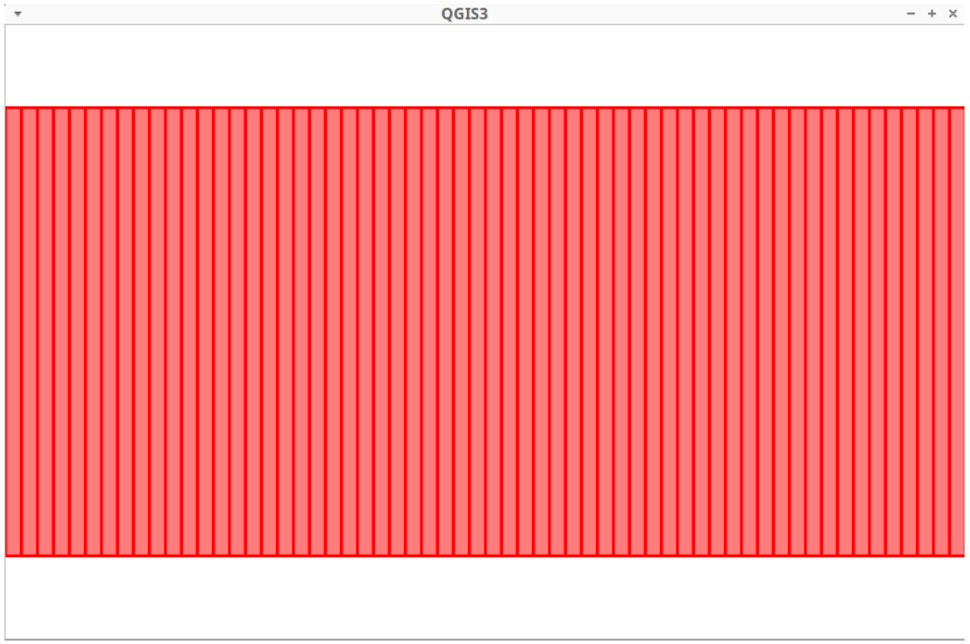


# 01 - CREATE AN UTM GRID

Write a script that generates an UTM grid. Allow the user to change the zone extend.

Input: the extent of the zone in degrees

Ouput:

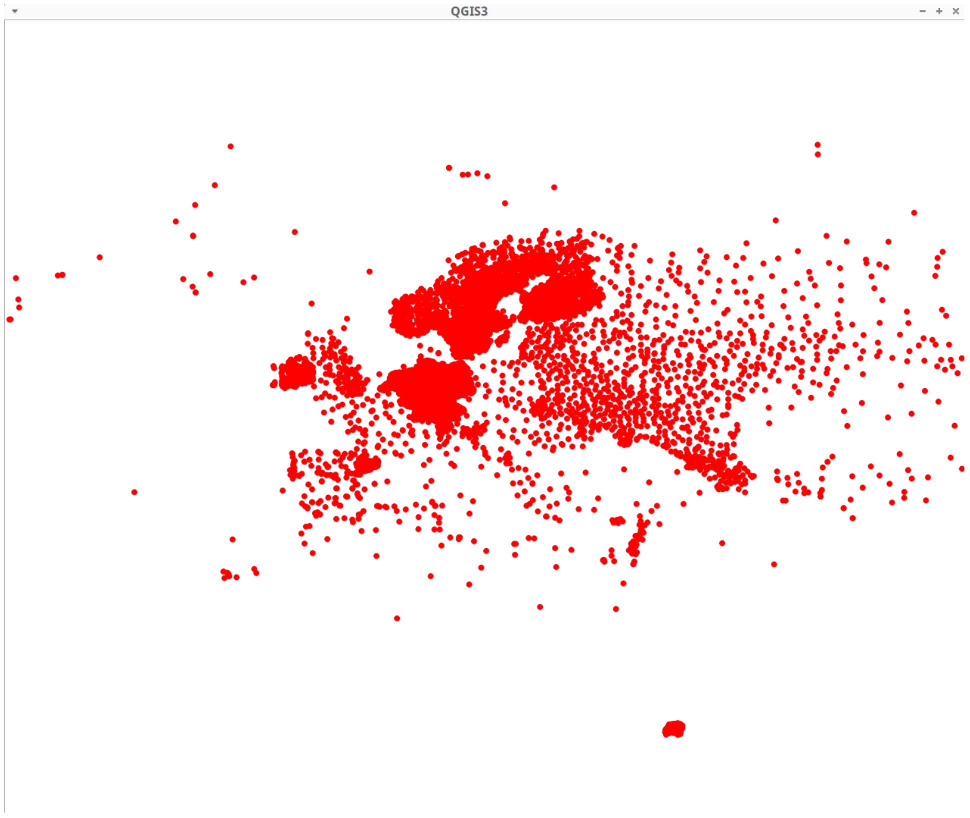


## 02 - PLOT STATIONS FILE

Write a script that plots the positions of the stations in the station file.

Input: the stations file

Ouput 1:



While at it, also print out the count of stations per country.

Output 2 (not necessarily sorted):

```
DE: 5597
SE: 1707
FI: 847
RU: 823
NO: 427
NL: 375
IE: 280
IT: 227
ES: 206
UA: 198
.
.
.
```

## 03 - FIND NEAREST STATION

Write a script that finds the station nearest to a given point.

Input:

- the check point coordinate
- the stations file

Ouput for the university building (11.34999, 46.49809):

```
JENESIEN  ->  Point (11.33000000000000007 46.53499999999999659)
```

## 04 - FIND STATIONS WITHIN RADIUS

Write a script that finds the stations within a radius in km of a given point using a point-buffering technique.

Input:

- the check point coordinate
- the radius in km
- the stations file

Output for the university building and radius 20Km:

```
FONDO ( 17 km) -> POINT(11.134999999999979 46.439999999999773)
JENESIEN ( 4 km) -> POINT(11.330000000000007 46.534999999999659)
KASTELRUTH ( 17 km) -> POINT(11.560000000000005 46.570000000000028)
MENDEL ( 14 km) -> POINT(11.205000000000007 46.419999999999946)
RADEIN ( 17 km) -> POINT(11.400000000000036 46.344999999999886)
ST.NIKOLAUS BEI KALTERN ( 12 km) -> POINT(11.234999999999943 46.419999999999946)
```

```
<license>
  This work is released under Creative Commons Attribution
  Share Alike (CC-BY-SA).
</license>
<sources>
  Much of the knowledge needed to create this training material has
  been produced by the sparkling knights of the
  <a href="http://www.osgeo.org">OSGEO</a> and
  <a href="http://qgis.org">QGIS</a>,
  communities.
  Their websites are filled up with learning material that can be used
  to grow knowledge beyond the boundaries of this lessons
  Another essential source has been the Wikipedia project.
</sources>
<important>
  This work is part of the Advanced Geomatics Course given in 2024 at the
  EMMA Master of the Free University of Bolzano.
</important>
```