documentation

May 23, 2024

1 LISTS

```
[ ]: mylist = ["Merano", "Bolzano", "Trento"]
print("The elements start at position 0: " + mylist[0])

mylist.append("Potsdam") # add an element at the end of list
mylist.remove("Potsdam") # remove element by object
mylist.pop(0) # remove element by index

doIHaveBolzano = "Bolzano" in mylist # check if an element is in a list, in␣
 ↪operator
print(doIHaveBolzano)
```

1.1 Looping over lists

If an index is necessary, we can use a “range” and the “len” function.

```
[ ]: colors = ["red", "green", "blue", "purple"]
ratios = [0.2, 0.3, 0.1, 0.4]

for index in range(len(colors)):
    ratio = ratios[index]
    color = colors[index]
    print(f"{color} -> {ratio}")
```

1.2 Break and continue

```
[ ]: for i in range(10): # break is used to exit a loop early
    if i == 5:
        break
print(f"A) {i}")

for i in range(10): # continue is used to skip rest of code inside loop␣
 ↪(current iteration)
    if i == 5:
        continue
print(f"B) {i}")
```

1.3 Ranges

Ranges produce a sequence of consecutive integers.

[ ]:
```
# Loop over a range form 0 to 10. Last number is not included.
for i in range(0,10):
    print(f"A) {i}")
```

[ ]:
```
# Initial value is optional
for i in range(10):
    print(f"B) {i}")
```

[ ]:
```
# Step can be set; here every second value
for i in range(0,10,2):
    print(f"C) {i}")
```

[ ]:
```
# Also in descending order
for i in range(10,0,-2):
    print(f"D) {i}")
```

1.4 Sorting lists

The sort method has a key parameter used to specify a function to be called on each list element prior to making comparisons.

[11]:
```
mylist = ["Merano", "Bolzano", "Trento"]
mylist.sort() # Alphabetically sorted
mylist.sort(reverse = True)
```

[ ]:
```
mylist = ["banana", "Orange", "Kiwi", "cherry"]
mylist.sort() # Sorts upper case first, then lowercase
mylist.sort(key = str.lower) # sorts alphabetically and ignores case
```

Lists can be merged with the plus operator. Lists can be concatenated to a string using separator.join(list). In case of numbers some functions apply.

[ ]:
```
abc = ["a", "b", "c"]
cde = ["c", "d", "e"]
newabcde = abc + cde

nums = [1.0, 2, 3.5]
print( max(nums) )
print( min(nums) )
print( sum(nums) )
```

2 DICTIONARIES

A Hashmap or Dictionary is a container of key and value pairs. Think of it as an actual dictionary, where you have definitions (the value) stored under certain names (the key). So you can ask the

dictionary for the definition of using the name. Mind that names/keys are case sensitive. Also keys are unique, so you can't have two definitions for the same name. If you insert a new value for an existing key, the old value is overwritten.

2.1 Create, Get, Add, Remove

[ ]:
```
townsProvinceMap = {
    "merano":"BZ", "bolzano":"BZ", "trento":"TN"
}

print(townsProvinceMap["merano"]) # Get a value through its key
townsProvinceMap["Potsdam"] = "BR" # Add a new key/value pair
townsProvinceMap.pop("Potsdam") # Remove by key/value

if townsProvinceMap.get("Merano") is None: # Key doesn't exist to avoid error
    print("The key doesn't exist")
else:
    print("The key exists")

print( townsProvinceMap.get("merano", "unknown") )
```

2.2 Looping Dictionaries

Remember that a dictionary item is a key/value pair, so we need 2 variables, but apart of that, looping is the same as for lists.

[ ]:
```
for key, value in townsProvinceMap.items():
    print( key + " is in province of " + value )
```

2.3 Keys and Values

In python, dictionaries are ordered following the insertion order. If sorting by key is needed, the best way to do so is to sort the keys and loop over them. Since the keys() method returns an iterable, we can't directly sort it (even if we can loop over it). We need to convert it to list first.

[ ]:
```
print( townsProvinceMap.keys() )
print( townsProvinceMap.values() )

towns = list(townsProvinceMap.keys()) # Convert dictionary to list to sort
towns.sort()
for town in towns:
    print( town + " is in province of " + townsProvinceMap[town] )
```

[ ]:
```
myText = """
We would like to know how many times
every character appears in this text.
"""
charDictionary = {} # New Dictionary
```

```
for character in myText.strip(): # strip removes whitespaces
    count = charDictionary.get(character, 0) # character as key
    count += 1
    charDictionary[character] = count # count as value

for key, value in charDictionary.items():
    print(key, "appears", value, "times.")
```

3 Text File

3.1 Writing a file

Defining a folder and file path.

[ ]:
```
folder = "C:/users/aschw/OneDrive - Scientific Network South Tyrol/" # Define folder path
filepath = f"{folder}/data.txt" # Define file path
```

[ ]:
```
data = """# station id, datetime, temperature
    1, 2023-01-01 00:00, 12.3
    2, 2023-01-01 00:00, 11.3
    3, 2023-01-01 00:00, 10.3"""

with open(filepath, "w") as file: # w stands for write
    file.write(data)

with open(filepath, "a") as file: # a appends to an existing file
    file.write("\n1, 2023-01-02 00:00, 9.3")
    file.write("\n2, 2023-01-02 00:00, 8.3")
```

3.2 Reading a file, parsing data

[ ]:
```
with open(filepath, "r") as file: # Read file
    lines = file.readlines()

stationCount = {}

for line in lines: # looping over file
    line = line.strip() # deleting whitespaces
    if line.startswith("#") or len(line) == 0: # comments and empty lines are ignored
        continue
    stationId = line.split(",")[0] # delimiter "," and variable at 1st position
    count = stationCount.get(stationId, 0) # get for unique values of stations
    count += 1
```

```
    stationCount[stationId] = count # count is assigned as new value to␣
 ↪stations (key)
```

4 PYQGIS

[ ]:
```
from pyqgis_scripting_ext.core import *
```

Define a Style

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

4.1 Creating Geometries

4.1.1 Create a point

[ ]:
```
point = HPoint(30.0, 10.0) # lon, lat --> Längengrad, Breitengrad --> X, Y
print(point.asWkt()) # print as well known text
```

4.1.2 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]] # pairs of floating point values

points = []
for coord in coords:
    points.append(QgsPoint(coord[0], coord[1]))

line = HLineString.fromCoords(coords) # line from coords
```

4.1.3 Create a polygon

A polygon is created using a closed linestring. This will be the exterior ring of the polygon.

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

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) # Create polygon from␣
 ↪exterior ring
```

```
holeRing = HLineString.fromCoords(holePoints) # Create hole ring
polygonWithHole.add_interior_ring(holeRing) # Add interior ring to polygon
print(polygonWithHole.asWkt())
```

4.2 Multi-Geometries

4.2.1 Multipoint

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

4.2.2 Multiline

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

4.2.3 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])

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

4.2.4 Subgeometries and Coordinates

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

[ ]:
```
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]}")
```

4.3 Create Geometries from WKT

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

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)
```

4.4 Check type of Variable

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

if isinstance(firstGeom, HPolygon): # as part of a script, against a class
    print(f"It indeed is a Polygon!")
```

5 MAP CANVAS, 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]])
```

[ ]:
```
canvas = HMapCanvas.new() # Define the new canvas

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() # Show output
```

5.1 Envelope of geometry, bbox

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

5.2 Length, area and distance

[ ]:
```
print("polygon length:", g1.length()) # perimeter
print("polygon area:", g1.area())

print("distance between line and point:", g5.distance(g4)) # planar distance␣
 ↪between two nearest points
```

5.3 Predicates

5.3.1 Intersects, touches, contains

[ ]:
```
print(g1.intersects(g2)) # Geometries that touch also intersect
print(g1.touches(g2)) # Geometries have at least one point in common; interiors␣
 ↪do not intersect
print(g1.contains(g2)) # Contain; not a border touch
```

5.4 Functions

5.4.1 Intersection, Symdifference, Combine, Difference, Buffer

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

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

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


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

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)
```

5.4.2 Convex hull

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

6 PROJECTIONS

[ ]:
```
crsHelper = HCrs() # Create the CRS helper
crsHelper.from_srid(4326) # from
crsHelper.to_srid(32632) # to

point4326 = HPoint(11, 46)
point32632 = crsHelper.transform(point4326) # Transform
backTo4326 = crsHelper.transform(point32632, inverse = True) # Reverse␣
 ↪transformation
```

7 LIFESAVER MAP

[ ]:
```
osm = HMap.get_osm_layer() # get OSM Layer
HMap.add_layer(osm) # Add layer to current map
HMap.remove_layers_by_name(["OpenStreetMap", "other map"]) # remove layers
```

8 READING AN EXISTING GPKG LAYER

[ ]:
```
folder = "....\" # Define folder path
geopackagePath = folder + "natural_earth_vector.gpkg"

countriesName = "ne_50m_admin_0_countries"
countriesLayer = HVectorLayer.open(geopackagePath, countriesName) # open

print("Schema (first 4 fields):")
counter = 0
for name, type in countriesLayer.fields.items():
    counter = counter + 1
    if counter < 5:
        print("\t", name, "of type", type)
```

[ ]:
```
crs = countriesLayer.prjcode
print("Projection: ", crs)
print("Spatial extent: ", countriesLayer.bbox())
print("Feature count: ", countriesLayer.size())
```

[ ]:
```
print("Attributes for Italy:")
countriesFeatures = countriesLayer.features() # Get features iterator
nameIndex = countriesLayer.field_index("NAME") # Index of field name
fieldNames = countriesLayer.field_names # Get all fields names
```

```
for feature in countriesFeatures:
    if feature.attributes[nameIndex] == 'Italy': # Access attributes by their␣
 ↪index
        geom = feature.geometry # get the geometry
        print("GEOM:", geom.asWkt()[:50] + "...")
        count = 0
        for index, attribute in enumerate(feature.attributes):
            print(fieldNames[index] + ":", attribute)
            count += 1
            if count > 5:
                print("...")
                break
```

9 FILTERS USING EXPRESSIONS

https://docs.qgis.org/3.28/en/docs/user_manual/expressions/functions_list.html

[ ]:
```
expression = "NAME like 'I%' and POP_EST > 30000000" # Filter features with "I"␣
 ↪and population
features = countriesLayer.features(expression) # Apply filter
count = 0 # Initialize count

for feature in features:
    print(feature.attributes[nameIndex])
    count+=1
```

9.1 BBOX Filter

A BBOX filter can be create using a QgsRectangle. This can be used for example to find cities within a "radius" of 200 km (~ 2 degrees) from Trento:

[ ]:
```
lon = 11.119982
lat = 46.080428
point = HPoint(lon, lat)
buffer = point.buffer(2)
citiesLayer = HVectorLayer.open(geopackagePath, citiesName)
HMap.add_layer(citiesLayer)

citiyNameIndex = citiesLayer.field_index("NAME")
print("\napply bbox filter on features")
aoi = buffer.bbox()
count = 0

for feature in citiesLayer.features(bbox=aoi): # filter by bbox; bbox as object␣
 ↪of features
    print(feature.attributes[citiyNameIndex])
```

```
    count += 1
```

9.2 Exact Geometry Filter

In pyQGIS there is no way to create an exact geometry filter. Therefore, to have an exact filter, two steps are necessary. Apply the bbox filter to the datasource (important for remote databases), and then check for intersection on the resulting features. The extension do that transparently for you:

[ ]:
```
for feature in citiesLayer.features(geometryfilter=buffer): # Filter by␣
 ↪geometry with geometry object as features method
    print(feature.attributes[citiyNameIndex])
```

10 IN-MEMORY VECTOR LAYER

One good way to start when creating new data, is the creation of a memory layer. This won't write any data on disk, until you tell it to do so. The first thing to define when creating a new dataset, is its schema, i.e. its fields and datatypes (possible types are string, integer, double). This is naturally done using a dictionary:

[ ]:
```
fields = {
    "id": "Integer",
    "name": "String",
}

just2citiesLayer = HVectorLayer.new("test", "Point", "EPSG:4326", fields) #␣
 ↪defined structure
just2citiesLayer.add_feature(HPoint(-122.42, 37.78), [1, "San Francisco"]) #␣
 ↪adding features
just2citiesLayer.add_feature(HPoint(-73.98, 40.47), [2, "New York"])
```

11 CREATE A NEW GEOPACKAGE

[ ]:
```
path = folder + "test.gpkg"
error = just2citiesLayer.dump_to_gpkg(path, overwrite=True) # Dump the layer to␣
 ↪geopackage
if(error):
    print(error) # Print out any error, if
HMap.add_layer(just2citiesLayer)
```

[ ]:
```
fields = {
    "name": "String",
    "population": "Integer",
    "lat": "double",
    "lon": "double"
    }
```

```
oneCityMoreAttributes = HVectorLayer.new("test2", "Point", "EPSG:4326", fields)
oneCityMoreAttributes.add_feature(HPoint(-73.98, 40.47), ["New York", 19040000,␣
 ↪40.47, -73.98])

path = folder + "test2.gpkg"

error = oneCityMoreAttributes.dump_to_gpkg(path, overwrite=True)

if(error):
    print(error)
```

12 STYLE

12.1 Point Style

[ ]:
```
citiesLayer.subset_filter("SOVONAME='Italy'") # Filter Italian cities
citiesLayer.set_style(pointStyle) # Apply style to layer

field = "NAME"
pointStyle += HLabel(field, yoffset = -5) + HHalo("red", 1) # Default Label
```

[ ]:
```
labelProperties = {
    "font": "Arial",
    "color": 'black',
    "size": 10,
    "field": field,
    "xoffset": 0.0,
    "yoffset": -5
    }

pointStyle += HLabel(**labelProperties) + HHalo("white", 1) # Adding labels to␣
 ↪pointStyle
```

12.2 Conditional labelling

[ ]:
```
if( condition, result if true, result if false)
field = "if(POP_MAX>1000000,concat(NAME,'(',round(POP_MAX/1000000,1),')'),NAME)"
```

12.3 Polygon Style

[ ]:
```
countriesLayer.subset_filter("NAME='Italy'")
polygonStyle = HFill('0,255,0,128') + HStroke('0,255,0,255', 2) # define␣
 ↪opacity and stroke
countriesLayer.set_style(polygonStyle)
```

12.4 Line Style with labelling

The rivers layer doesn't have an attribute to extract the rivers in Italy. We will need an intersection filter. The sub_layer function allows to extract a sublayer based on a geometry filter. It is also possible to define which fields of the original layer to keep.

[ ]:
```
riversItalyLayer = riversLayer.sub_layer(italyFeatures[0].geometry,␣
 ↪"rivers_italy", ['scalerank', 'name'])
riversStyle = HStroke('0,0,255,255', 2)

riversItalyLayer.set_style(riversStyle)

labelProperties = {
    "font": "Arial",
    "color": 'red',
    "size": 14,
    "field": 'name',
    "along_line": True, # Label follows direction of line
    "bold": True,
    "italic": False
}
labelStyle = HLabel(**labelProperties) + HHalo("white", 1)
riversStyle = HStroke('0,0,255,255', 2) + labelStyle

HMap.add_layer(riversItalyLayer)
```

12.5 Advanced styling, graduated

[ ]:
```
ranges = [
    [0, 0],
    [1, 5],
    [6, 8],
    [8, 9],
    [10, 11]
]
styles = [
    HStroke('0,0,255,255', 7),
    HStroke('0,0,255,255', 5),
    HStroke('0,0,255,255', 3),
    HStroke('0,0,255,255', 2),
    HStroke('0,0,255,255', 1),
]

riversItalyLayer.set_graduated_style('scalerank', ranges, styles, labelStyle)
```

13 PRINTING DATA TO IMAGE AND PAPER

[ ]:
```
printer = HPrinter(iface)

mapProperties = {
    "x": 5, # Define paper position and size in mm
    "y": 25,
    "width": 285,
    "height": 180,
    "frame": True, # Add a frame around the map and add map object to layout
    "extent": [10, 44, 12, 46] # Set extent
}
printer.add_map(**mapProperties)

labelProperties = {
    "x": 120,
    "y": 10,
    "text": "Such a nice map!",
    "font_size": 28,
    "bold": True,
    "italic": False
}
printer.add_label(**labelProperties) # Add label to the map

legendProperties = {
    "x": 215,
    "y": 30,
    "width": 150,
    "height": 100,
    "frame": True,
    "max_symbol_size": 3
}
printer.add_legend(**legendProperties) # Add legend to the map

scalebarProperties = {
    "x": 10,
    "y": 190,
    "units": "km",
    "segments": 4,
    "unit_per_segment": 10,
    "style": "Single Box", # or 'Line Ticks Up'
    "font_size": 12
}
printer.add_scalebar(**scalebarProperties) # Add scalebar
```

[ ]:
```
path = ".\test.pdf" # Print to PDF
printer.dump_to_pdf(path)
```

```
path = ".\test.png" # Print to png
printer.dump_to_image(path)
```