Szkriptek és pluginek fejlesztése QGIS-hez

Dr. Ungvári Zsuzsanna
adjunktus
ELTE IK Térképtudományi és Geoinformatikai Intézet

Lektorálta:

Dr. Kerkovits Krisztián András

egyetemi docens

Tartalomjegyzék

Előszó 2
Egyéb, ajánlott olvasmányok, segédanyagok 2
Bevezetés: a QGIS Python konzol 3
Alapfogalmak 3
Vektoros adatok kezelése Pythonnal 5
Fájlbetöltés, rétegkezelés 5
Stílusok kezelése 7
Hozzáférés a réteg attribútum táblázatához, kijelölések 20
Az attribútum táblázat kezelése PyQGIS-ben 20
Lekérdezések és kijelölések 22
Attribútum adatok szerkesztése 24
Műveletek az elemek geometriájával: létrehozás, módosítás, elem törlése 28
Új réteg létrehozása 34
Vetületek kezelése 36
A feldolgozó eszközök 37
Raszteres képek olvasása, pixelértékek megszerzése 41
Raszteres képe írása, műveletek raszteres fájlokkal 45
Összetett feladatok vektoros és raszteres adatokkal 47
A QGIS visszacsatoló üzenetei 59
A pluginfejlesztés kezdő lépései 62
Néhány gondolat a pluginfejlesztésről 62

Előszó

Ez a jegyzet két fontosabb témával foglalkozik: az egyik a szkriptek írása a QGIS Python konzolban, a másik pedig a modulok fejlesztése QGIS-hez. Ahhoz, hogy ezt el tudjuk kezdeni, bizonyos előismeretekre van szükség, amelyek a Geoinformatika mesterszakon a következő tárgyakból kerülnek ki:

- Adatbányászat, felhő alapú adatok

- Algoritmusok a geoinformatikában

- Vektoros térinformatika (QGIS)

→ vagyis jó Python alapok és QGIS haladó szintű ismerete szükséges!

Egyéb, ajánlott olvasmányok, segédanyagok

PyQGIS Developer Cookbook: A fejlesztői dokumentáció megismerése fontos folyamat bármilyen fejlesztés megkezdése előtt. Jelenlegi legfrissebb verzió elérhető itt:

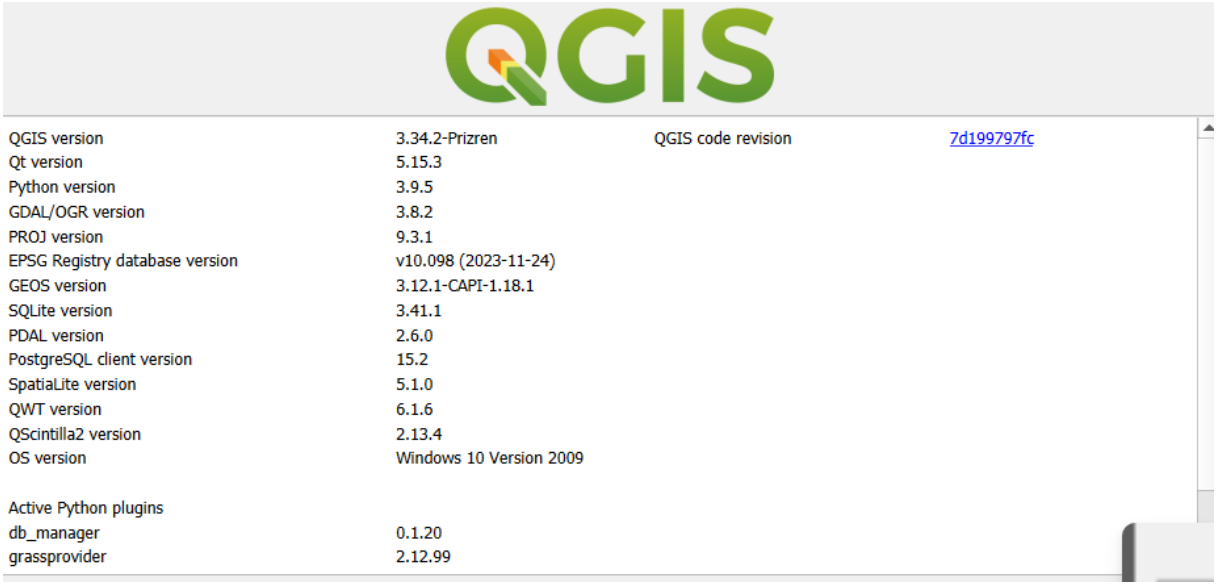
https://docs.qgis.org/3.34/en/docs/pyqgis_developer_cookbook/index.html

Korábbi verziókhoz készült segédletek egyébként PDF-ben is elérhetők.

Valamint a legfrissebb QGIS Python API dokumentáció itt érhető el:

https://qgis.org/pyqgis/master/

A jelenlegi szkripteket és plugineket a QGIS aktuálisan legújabb verzióján teszteltem, ez a 3.34-es LTR verzió. Az általam használt QGIS környezet verziószámai az alábbi képen olvashatók.



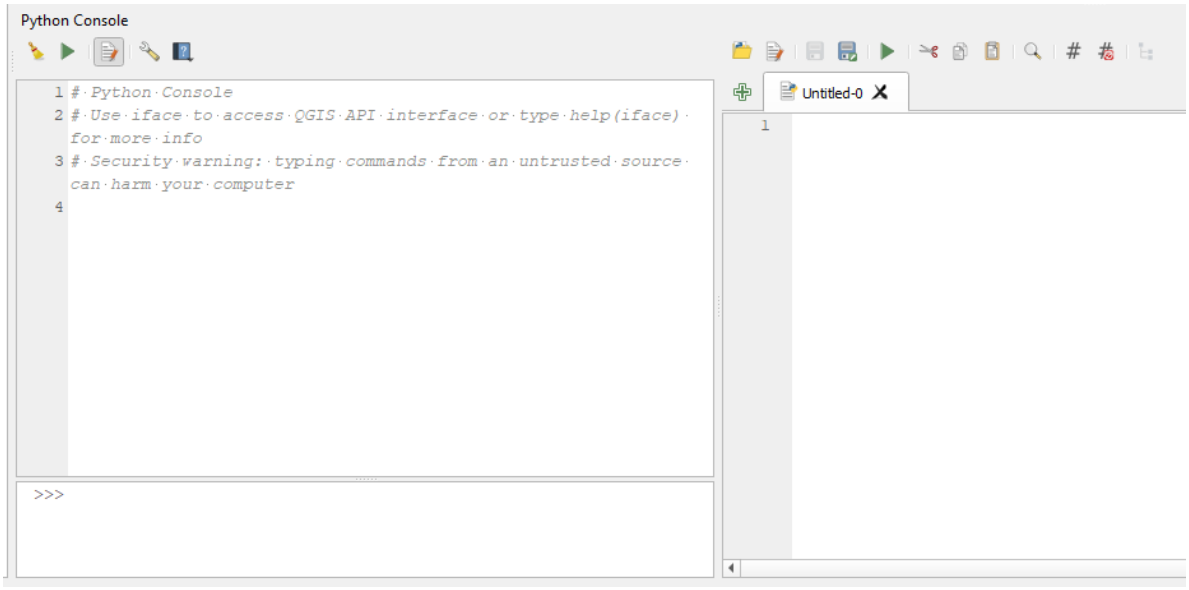
A jegyzet melléklete a bemutatott programkódok és az adatok.

Bevezetés: a QGIS Python konzol

A QGIS – mint más térinformatikai szoftverek is – lehetőséget kínál arra, hogy a felhasználók saját maguk írjanak kiegészítő modulokat, programkódokat, amelyek nincsenek benne az alapszoftverben, esetleg szabadon kombinálhassák a szoftver meglévő eszközeit saját céljuk végrehajtására. A QGIS 0.9-es verziója óta tudunk Python szkripteket írni a programhoz (2007): erre azért volt szükség, mivel a C++ nyelvhez értők száma jóval kevesebb (ezen a nyelven is írható a kód), így egy átlagos felhasználó nem vagy csak hosszas tanulás árán tudna szkriptet vagy plugint fejleszteni a szoftverhez.

Ahhoz, hogy ezt meg tudjuk tenni, először is meg kell ismerni a szoftvert, illetve nem hátrány, ha a felhasználó rendelkezik alapvető Python programozási ismeretekkel. Ezen tudás birtokában kezdhető meg a szkript, illetve a pluginfejlesztés. A jegyzet első részében a szkriptek fejlesztésére készíti fel az olvasót. A jegyzet második része foglalkozik a pluginfejlesztéssel.

A QGIS 3.X-es verziói már a Python 3-as verziót használják. A Python konzol megnyitása a *Plugins* → *Python Console* menüben lehetséges.



A baloldalon látható a Python konzol, a zöld nyíllal futtatható le a program. Az első, seprűt ábrázoló ikon a konzol eddigi tartalmát törli. A papírlap ikonnal nyitható meg a szerkesztő, ahol menthetjük, illetve betölthetjük a szkripteket. Működése hasonló bármely más Python kódszerkesztőhöz.

Alapfogalmak

GUI: Graphical User Interface. Grafikus felhasználói felület.

Qt/PyQt: Egy Python API interfész a Qt-hez. (a Qt egy C++-ban íródott keretrendszer, amelyet a QGIS is használ). A PyQt ennek a Qt programnak egyszerűsített formája, ebben lehetőségünk van megtervezni a pluginek grafikus felületét. A jelenlegi verziókban PyQt a QGIS-szel együtt települ a gépre.

SIP: A SIP egy olyan eszközgyűjtemény, amely megkönnyíti a C és C++ könyvtárakhoz Python bináris kódok létrehozását (angolul Python bindings). Eredetileg 1998-ban fejlesztették ki a PyQt, a Qt eszközkészlet Python bináris kódjainak létrehozására, de bármilyen C vagy C++ könyvtárhoz készíthetünk vele bináris kódokat.

A SIP modul több építőeszközből áll. Ezek az építőeszközök feldolgozzák a specifikációs fájlokat, és C vagy C++ kódot generálnak belőlük, amelyet aztán lefordítanak a Python bináris

kódjainak létrehozásához. A specifikációs fájlok tartalmazzák a C vagy C++ könyvtár interfészének leírását, azaz az osztályokat, metódusokat, függvényeket és változókat. A specifikációs fájl formátuma szinte teljesen megegyezik a C vagy C++ fejlécfájléval. Egy Python-csomagba több bővítőmodul is telepíthető. A bővítőmodulok úgy építhetők, hogy függetlenek legyenek a Python használt verziójától. Más szóval a belőlük létrehozott eszköz a Python bármelyik verziójával telepíthető a v3.8-tól kezdve.

A **renderelés**: a szoftveres renderelés olyan grafikus megjelenítési, illetve leképezési mód, amiben a program (tipikusan játékok motorja) a grafikát teljes mértékben szoftveres úton állítja elő, és már csak a kész képet adja át megjelenítésre a videóvezérlőnek.

A legfontosabb QGIS könyvtárak

QGIS CORE Library: Minden alapvető QGIS függvényt, eszközt tartalmazó függvénykönyvtár. Ide tartozik minden olyan eszköz, amelyik nem a felhasználói felület beállításaihoz tartozik.

QGIS GUI Library: Ez a függvénykönyvtár vezérli a grafikus felületet, vagyis tartalmaz minden olyan függvényt, amely a felhasználói felület beállításait végzi el, pl. dialógusablakok, widgetek.

QGIS UTILS Library: Azon eszközök csoportja, amelyet a QGIS a Python alapú dolgok elvégzéséhez használ. Pl. Python hibakezelés hozzáadása stb.

Az iface(): az iface változó a QgisInterface osztály egy példánya. Az interfész hozzáférést biztosít a menükhöz, toolbarokhoz, a térképi vászonhoz, valamint a QGIS egyéb részeihez.

A providerekről

A QGIS-ben sokféle adatformátumot tudunk írni és olvasni egyaránt. Ezekért a providerek (driverek, meghajtók) felelnek. A konzolban a következő kóddal kilistázhatjuk a drivereket.

```
for provider in QgsProviderRegistry.instance().providerList():
    print (provider)
```

A providereket az alábbi lista mutatja be. Vastag betűvel kiemeltem a számunkra legfontosabbakat.

OAPIF	delimitedtext	memory	tiledscene
WFS	ept	mesh_memory	vectortile
arcgisfeatureserver	**gdal**	mssql	virtual
arcgismapserver	gpx	**ogr**	virtualraster
arcgisvectortileservice	grass	oracle	vpc
cesiumtiles	grassraster	pdal	vtpkvectortiles
copc	hana	postgres	wcs
	mbtilesvectortiles	postgresraster	wms
	mdal	spatialite	xyzvectortiles

Az ogr driver felel az általános vektoros formátumok megnyitásáért, így például a Shapefile, és a Geopackage. A raszteres fájlok közül a GeoTIFF-ek olvasását a gdal végzi.

https://qgis.org/pyqgis/master/core/QgsVectorLayer.html

Vektoros adatok kezelése Pythonnal

Fájlbetöltés, rétegkezelés

A feladatokhoz szorosan kapcsolódik a mellékelt adatsor, amelyből dolgozhat a hallgató, valamint az itt bemutatott szkripteket is tartalmazza a segédlet.

1.a feladat: Töltsünk be vektoros réteget, először egy Shapefile-t! 01_loading_shp_gpkg.py

Az újabb QGIS Python konzolban a *core* és a *gui* könyvtár már alapértelmezetten importálva van, ezért nem szükséges kiadnunk ezt az utasítást. Azonban, ha mégis szeretnénk megtenni, így lehetséges:

```
from qgis.core import *
from qgis.gui import *
```

Adjuk meg a fájl elérési útját (abszolút hivatkozást használva).

```
path="C:/adatok/könyvtára/megye.shp"
```

Egy réteg betöltése

Helyezzük egy változóba a réteget. A *core* osztályban található a *QgsVectorLayer* függvény. Ebben három paramétert adunk meg: az elérési utat, a réteg nevét a QGIS-ben, és a providert (gyakorlatilag a drivert, amivel beolvassuk a fájlt). A Shapefile-ok és a Geopackage-ek vektoros adatok, amelyhez az ogr drivert kell használni. A réteg példányosításához az *addMapLayer* függvényt fogjuk meghívni. Ez a függvény a *QgsProject.instance()* osztály része.

Megvizsgálhatjuk azt is, hogy a réteg létező adatforrás-e, ezt egy elágazással tegyük meg az *isValid()* függvénnyel:

```
megye=QgsVectorLayer(path,"megye","ogr")
if not megye.isValid():
    print("Layer failed to load!")
else:
    QgsProject.instance().addMapLayer(megye)
```

Létezik rövidebb módszer is arra, hogy behívjunk egy réteget. Ezt a réteg fizikai helyének megadás után az interfész *addVectorLayer()* metódusával tehetjük meg. A továbbiakban az egyszerűsége miatt inkább ezt fogom alkalmazni.

```
iface.addVectorLayer(path, "megye","ogr")
```

1.b feladat: Töltsünk be vektoros réteget, most egy Geopackage fájlt! 01_loading_shp_gpkg.py

A fent bemutatott első módszerrel a következőképpen fog kinézni. Az egyetlen különbség az adatforrás megadása. Mivel a *gpkg* egy adatbázisfájl, amely több rétegből állhat, ezért a réteg nevét is meg kell határozni a *layername* paraméterrel.

```
gpkg_megye=" C:/adatok/könyvtára/mo.gpkg|layername=megye"
megye = QgsVectorLayer(gpkg_megye, "megye", "ogr")
if not megye.isValid():
    print("Layer failed to load!")
else:
    QgsProject.instance().addMapLayer(megye)
```

Az *iface addMapLayer()* függvénye itt is működik, a megfelelő adatforrás megadással.

```
gpkg_megye=" C:/adatok/könyvtára/mo.gpkg|layername=megye"
iface.addVectorLayer(gpkg_megye, "megye","ogr")
```

Ha éppen több réteg van behívva, előfordul, hogy szeretnénk váltani az aktív rétegek között. A réteg neve ismeretében ezt így tudjuk megtenni.

```
layer=QgsProject.instance().mapLayersByName("réteg neve")[0]
```

Lekérdezzük a projektbe betöltött rétegeket a *mapLayersByName()* függvénnyel, majd az aktív réteggé tesszük: *[0]*.

Stílusok kezelése

2. feladat: Betöltés után változtassuk meg a réteg színét!

02_change_color.py

A megye poligon réteg betöltése után váltsuk át a poligonok színét vörösre.

```
path=" C:/adatok/könyvtára/megye.shp"
iface.addVectorLayer(path, "megye","ogr")
```

Ehhez először a réteget aktívvá kell tennünk, majd meg kell hívnunk rajta a renderelőt (*renderer()*). A renderelő felelős az egyes elemek (*feature*-ök) szimbólummal történő megjelenítéséért. Ez lehet a *single symbol*, a *categorized*, *graduated* és *rule-based* renderelő. (Ez a kód ekvivalens azzal, ha megnyitnánk a *Layer Properties* ablakot, és kiválasztanánk a *Symbology* almenüt).

```
active_layer=iface.activeLayer()
renderer=active_layer.renderer()
symbol=renderer.symbol()
```

A szimbólum színét többféleképpen is megadhatjuk, erre négy módszert is bemutatok, értelemszerűen, ebből elegendő az egyiket kiválasztani.

```
symbol.setColor(QColor('red')) #Angol névvel
symbol.setColor(QColor('#ff0000')) #Hexadecimális kóddal
symbol.setColor(QColor(255,0,0,255)) #RGBA értékkel
symbol.setColor(QColor(Qt.red))  #Qt könyvtár színeivel
```

Miután megvan a szín beállítása, alkalmazni kell az aktív rétegre a beállításokat (*triggerRepaint()*), valamint a rétegkezelőben is célszerű frissíteni az új szimbólumot. Először hozzáférünk az aktív réteghez a rétegkezelőben (*iface.layerTreeView()*), majd frissítjük a nézetet (*refreshLayerSymbology(aktív réteg id-ja)*).

```
active_layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(active_layer.id())
```

3. feladat: Módosítsuk a pontréteg stílusbeállításait!

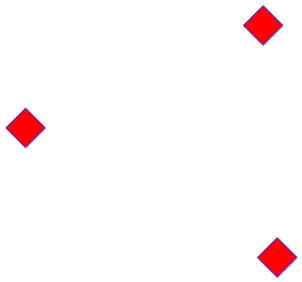
03_change_marker_symbol.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, pontokat tartalmazó réteg (a célnak megfelel a *mo.gpkg* település rétege).

Hozzunk létre egy *symbol* változót (*QgsMarkerSymbol()*), és a *createSimple* metódussal szabjuk át a tulajdonságait. Majd definiáljuk a renderelőt, és állítsuk be a *setSymbol()* függvénnyel az új szimbólumot. Ahogy az előző példában már láthattuk, itt is meg kell hívni a réteg és a rétegkezelő frissítését is.

```
layer=iface.activeLayer()
symbol = QgsMarkerSymbol().createSimple({'angle': 45,'color':
'red','name':'square','size':5, 'outline_width':0.2, 'outline_color':'blue'
})
renderer=layer.renderer()
renderer.setSymbol(symbol)
layer.triggerRepaint()
layer_tree=iface.layerTreeView()
```

```
layer_tree.refreshLayerSymbology(layer.id())
```



A következő táblázat foglalja össze, milyen tulajdonságokat tudunk változtatni a *createSimple()* függvénnyel.

Kulcs	Érték
angle	a marker elforgatásának szöge fokban
color	szín
horizontal_anchor_point	a marker vízszintes középpontjának megadása egész szám értékkel
name	a marker típusa angol megnevezéssel (négyzet: square, kör: circle stb.)
offset	az eredeti középponthoz képesti eltolás, x,y sorrendben: '5,3'
offset_unit	az eltolás mértékegysége: mapunit vagy mm
outline_color	körvonal színe
outline_width	körvonal vastagsága
outline_width_unit	körvonal vastagságának mértékegysége mapunit vagy mm
size	méret
size_unit	a méret mértékegysége mapunit vagy mm
vertical_anchor_point	a marker függőleges középpontjának megadása egész szám értékkel

A renderelést kétféleképpen tudjuk kivitelezni. Egyszer így:

```
renderer=layer.renderer()
renderer.setSymbol(symbol)
```

Másképpen pedig így:

```
renderer = QgsSingleSymbolRenderer(symbol)
```

```
layer.setRenderer(renderer)
```

A kettő kiváltja egymást. Az utóbbinál jobban megfigyelhető, hogy azt a renderelőt hívjuk meg, amelyik a rétegen minden elemet ugyanazzal a szimbólummal jelenít meg (*QgsSingleSymbolRenderer()*).

4. feladat: Módosítsuk a vonalas réteg stílusbeállításait!

04_change_line_symbol.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, vonalakat tartalmazó réteg (a célnak megfelel a *mo.gpkg* főutak rétege).

Hozzunk létre egy *symbol* változót (*QgsLineSymbol()*), és a *createSimple()* metódussal szabjuk át a tulajdonságait. Majd definiáljuk a renderelőt, és állítsuk be a *setSymbol()* függvénnyel az új szimbólumot. Ahogy az előző példában már láthattuk, itt is meg kell hívni a réteg és a rétegkezelő frissítését is.

```
layer=iface.activeLayer()
symbol = QgsLineSymbol().createSimple({'capstyle':'flat','color':
'gray','customdash': '3;3','penstyle':'solid','width':1,
'use_custom_dash':1 })
renderer=layer.renderer()
renderer.setSymbol(symbol)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

A következő táblázat foglalja össze, milyen tulajdonságokat tudunk változtatni a *createSimple()* függvénnyel.

Kulcs	Érték
capstyle	vonalvég stílusa: ’square’, ’flat’ vagy ’round’
color	szín
customdash	saját szaggatási módszer beállítása: ’dash;space’ vagyis ’5;3’
customdash_unit	a szaggatás mértékegysége mapunit vagy mm
joinstyle	a vonal törésének jellege: ’bevel’, ’miter’ vagy ’round’

offset	az eredeti középponthoz képesti eltolás, x,y sorrendben: '5,3'
offset_unit	az eltolás mértékegysége: mapunit vagy mm
penstyle	a vonal minősége. ’no’, ’solid’, ’dash’, ’dot’, ’dash dot’, ’dash dot dot’
use_custom_dash	használjuk-e az egyéni szaggatást, igen: 1
width	a vonal vastagsága
width_unit	a vonal vastagságának mértékegysége mapunit vagy mm

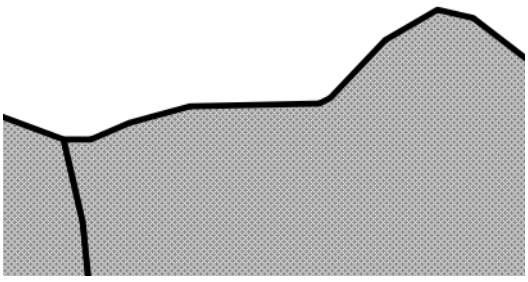
5. feladat: Módosítsuk a felületek stílusbeállításait!

05_change_fill_symbol.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, felületeket tartalmazó réteg (a célnak megfelel a *mo.gpkg* megyék rétege).

Hozzunk létre egy *symbol* változót (*QgsFillSymbol()),* és a *createSimple()* metódussal szabjuk át a tulajdonságait. Majd definiáljuk a renderelőt, és állítsuk be a *setSymbol()* függvénnyel az új szimbólumot. Ahogy az előző példában már láthattuk itt is meg kell hívni a réteg és a rétegkezelő frissítését is.

```
layer=iface.activeLayer()
symbol = QgsFillSymbol().createSimple({'color': 'gray','color_border':
'black','style':'dense3','width_border':1 })
renderer=layer.renderer()
renderer.setSymbol(symbol)
layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```



A következő táblázat foglalja össze, milyen tulajdonságokat tudunk változtatni a *createSimple()* függvénnyel.

Kulcs	**Érték**
border_width_unit	a körvonal vastagságának mértékegysége mapunit vagy mm

color	kitöltés színe
color_border	körvonal színe
offset	az eredeti középponthoz képesti eltolás, x,y sorrendben: '5,3'
offset_unit	az eltolás mértékegysége: mapunit vagy mm
style	kitöltés minősége: 'solid', 'horizontal', 'vertical', 'cross', 'b_diagonal', 'f_diagonal', 'diagonal_x', 'dense1'…'dense7'
style_border	körvonal minősége: 'no', 'solid', 'dash', 'dot', 'dash dot', 'dash dot dot'
width_border	körvonal vastagsága

6. feladat: Szimbólumok több réteggel. Készítsünk összetett pontjelet! Az előbbi élére állított piros négyzetbe helyezzünk el egy nála kisebb fekete pontot.

06_symbol_layers_marker.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, pontokat tartalmazó réteg (a célnak megfelel a *mo.gpkg* települések rétege).

Férjünk hozzá az aktív réteghez. Hozzunk létre egy pontszerű jelet (*QgsMarkerSymbol()*) a *createSimple()* metódussal. Ehhez adjunk hozzá egy új szimbólumréteget a *QgsSimpleMarkerSymbolLayer.create()* függvénnyel. Ahhoz, hogy ténylegesen is hozzá tudjuk adni a szimbólumon az *appendSymbolLayer(új réteg neve)* függvényt kell meghívni.

Ezután hívjuk meg a renderelőt, amely ebben az esetben (mivel minden elem a rétegen ugyanolyan szimbólummal jelenik meg) a *SingleSymbolRenderer().*

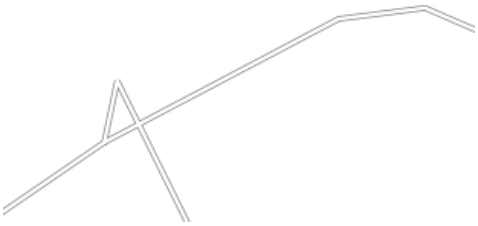
```
layer = iface.activeLayer()
symbol =QgsMarkerSymbol.createSimple ({'scale_method':'area','angle':
45,'color': 'red','name':'square','size':5, 'outline_width':0.2,
'outline_color':'blue' })
symbol_l2 = QgsSimpleMarkerSymbolLayer.create ({'color':'black',
'name':'circle', 'size':'2'})
symbol.appendSymbolLayer (symbol_l2)

#renderer = QgsSingleSymbolRenderer (symbol)
#layer.setRenderer(renderer) #vagy
renderer=layer.renderer()
```

```
renderer.setSymbol(symbol)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

7. feladat: Szimbólumok több réteggel. Készítsünk összetett vonalas jelet! A szimbólum legyen egy kétvonalas út.

07_symbol_layers_line.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, vonalakat tartalmazó réteg (a célnak megfelel a *mo.gpkg* főutak rétege).



Férjünk hozzá az aktív réteghez. Hozzunk létre egy vonalas jelet (*QgsLineSymbol()*) a *createSimple()* metódussal. Ehhez adjunk hozzá egy új szimbólumréteget a *QgsSimpleLineSymbolLayer.create()* függvénnyel. Ahhoz, hogy ténylegesen is hozzá tudjuk adni a szimbólumon az *appendSymbolLayer(új réteg neve)* függvényt kell meghívni.

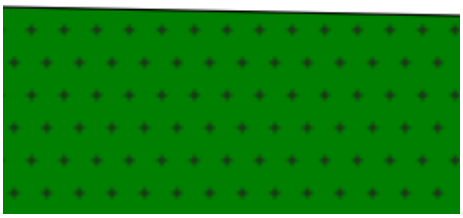
Ezután hívjuk meg a renderelőt, amely ebben az esetben (mivel minden elem a rétegen ugyanolyan szimbólummal jelenik meg) a *SingleSymbolRenderer()*. Lehetőség van a vonalaknál alkalmazni a szimbólumszinteket (*Advanced→Symbol Levels*), ezt kapcsoljuk be, hogy a vonalvégek eltűnjenek, folytonos vonalakat kapjunk. Erre a *setUsingSymbolLevels(True)* függvényt használjuk. Ezek után a nézet, és a rétegkezelő frissítése marad hátra a szokásos módon

```
layer = iface.activeLayer()
symbol =QgsLineSymbol.createSimple ({'color':'gray', 'width':'0.8',
'penstyle':'solid'})
symbol_l2 = QgsSimpleLineSymbolLayer.create ({'color':'white',
'width':'0.5', 'penstyle':'solid'})
symbol.appendSymbolLayer (symbol_l2)

renderer = QgsSingleSymbolRenderer(symbol)
renderer.setUsingSymbolLevels(True)
layer.setRenderer(renderer)
layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

8. feladat: Szimbólumok több réteggel. Készítsünk összetett felületi jelet! A felület legyen kitöltve zöld színnel, rajta a szimbólum legyen keresztek sokasága, amely a temetőt szimbolizálja.

08_symbol_layers_fill.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, felületeket tartalmazó réteg (a célnak megfelel a *mo.gpkg* megye rétege).



Férjünk hozzá az aktív réteghez. Hozzunk létre egy kitöltésszimbólumot (*QgsFillSymbol*). Majd adjunk hozzá egy újabb szimbólumréteget, ez a *QgsPointPatternFillSymbolLayer()* a pontszerű mintázat kitöltést. Ennek a *create()* beállításainál csak a mintázat beállításait lehet megtenni (úgy mintha a *Layer Properties* ablak *Symbology* menüjében a *Point pattern fill* szintjén állítanám a mintázatot). A mintázat sűrűségét és a sorok/oszlopok egymáshoz való „kicsúsztatását" állítom be. A szimbólum beállításait (méret, szín) függvényekkel is változtatható, viszont maga a szimbólum alakja nem. Éppen ezért kitöröltem a marker szimbólum rétegét, és újra létrehoztam (*QgsSimpleMarkerSymbolLayer().create())* paranccsal.

Ezek után a szokásos renderelőt és a nézet és rétegkezelő frissítését kell meghívni.

```
layer = iface.activeLayer()
symbol =QgsFillSymbol.createSimple ({'color': 'green','color_border':
'black' })
symbol_l2 = QgsPointPatternFillSymbolLayer.create
({'distance_x':4,'distance_y':4, 'displacement_x':2})
subSymbol = symbol_l2.subSymbol()

subSymbol.deleteSymbolLayer(0)
cross =QgsSimpleMarkerSymbolLayer().create ({'color': 'black',
'name':'cross','size':1 })
subSymbol.appendSymbolLayer(cross)
symbol.appendSymbolLayer (symbol_l2)

renderer=layer.renderer()
renderer.setSymbol(symbol)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

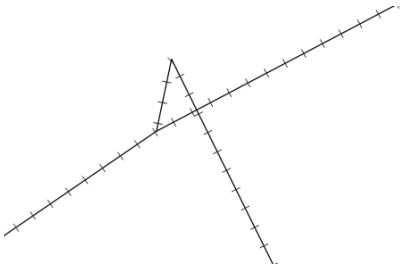
Ha a szimbólum színét vagy méretét állítanám át, megfelelő lenne a következő formula is:

```
symbol_l2.setColor(QColor('black'))
symbol_l2.subSymbol().setSize(1)
```

9. feladat: Készítsünk egy marker line szimbólumot. A szimbólum legyen egy vasútjel, folytonos vékony fekete vonal, rajta merőlegesen egyenes rövid vonalazás.

09_marker_line_symbol.py

A feladat előfeltétele, hogy a QGIS-ben legyen betöltve bármilyen, vonalakat tartalmazó réteg (a célnak megfelel a *mo.gpkg* vasutak rétege).



Férjünk hozzá az aktív vonalas réteghez. Hozzunk létre egy vonalas jelet a *QgsLineSymbol.createSimple()* függvénnyel. Ezek után ehhez a szimbólumhoz adjunk hozzá egy új szimbólumréteget, egy *Marker Line*-t: *QgsMarkerLineSymbolLayer.create ({'interval':5 }).* Ezen a szinten csak a *Marker Line* markereinek sűrűségét tudjuk megadni az *interval* paraméterrel. Majd az előző példához hasonlóan töröljük ki a jelenlegi szimbólumot, és hozzunk létre egy újat (*QgsSimpleMarkerSymbolLayer()*) függvénnyel, méghozzá egy rövid vonalszimbólumot (ez a *line*).

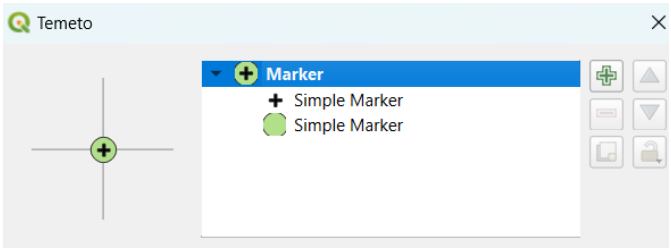
Ezek után a szokásos renderelőt és a nézet és rétegkezelő frissítését kell meghívni.

```
layer = iface.activeLayer()

symbol =QgsLineSymbol.createSimple ({'color':'black','width:':0.5 })
symbol_l2 = QgsMarkerLineSymbolLayer.create ({'interval':5 })
subSymbol = symbol_l2.subSymbol()

subSymbol.deleteSymbolLayer(0)
line =QgsSimpleMarkerSymbolLayer().create ({'color': 'black',
'name':'line','size':2 })
subSymbol.appendSymbolLayer(line)
symbol.appendSymbolLayer (symbol_l2)

renderer=layer.renderer()
renderer.setSymbol(symbol)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

10. feladat: Állítsunk be egy meglévő (elmentett szimbólumot) az éppen behívott rétegnek!

10_default_style.py

Állítsunk be egy stílust, legyen az *mo.gpkg* település rétegéhez egy új szimbólum, pl. egy temetőszimbólum, vagyis egy zöld körben egy kereszt.



Mentsük el az új stílust, mint QML fájl. A QML fájl egy XML típusú fájl, amelyikben szövegesen tárolja a réteghez kapcsolódó stílusokat a QGIS. A mentéshez a réteg nevén jobb egérgombbal kattintsunk → *Export* → *Save as QGIS Layer Style File*. A menüben állítsuk be az *As QGIS QML style file* lehetőséget, és adjuk meg a mentés helyét. A menü alsó részében megadható, hogy a réteg mely tulajdonságait mentsük el, pl. szövegek, diagramok stb.

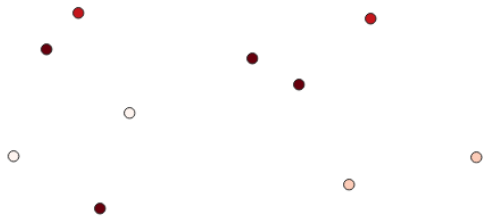
Ha ez megvan, a *loadNamedStyl(elérési út)* függvényt kell használnunk a réteg új stílusának betöltéséhez.

```
layer=iface.activeLayer()
layer.loadNamedStyle('C:/adatok/forrása/temeto.qml')
layer.triggerRepaint()
iface.layerTreeView().refreshLayerSymbology(layer.id())
```

11. feladat: Készítsünk kategorizált ábrázolást a település réteghez jogállás alapján!

11_categorized_styles.py

Soroljuk csoportokba a jogállás mező különböző értékei alapján a településeket. A jelek mérete és formája azonos. A színe az egyik, pl. vörös színskála szerint változik.



Miután megvan az aktív réteg, definiáljuk, hogy az attribútum táblázat melyik mezője alapján szeretnénk csoportokba sorolni az elemeket. Ehhez hozzá kell férni a rétegen található mezők tartalmához, méghozzá a jogálláshoz. Ennek az indexét (hányadik oszlop a rétegen) kérdeztem le. Majd a réteg különböző értékeit a *uniqueValues()* metódussal szerzem meg, ezek az értékek egy listába kerülnek.

```
layer = iface.activeLayer()

categories=[]
f="jogallas"
fni = layer.dataProvider().fields().indexFromName(f)
unique_values = layer.uniqueValues(fni)
```

Ezután a marker réteg *default* szimbólumát (pont) fogom használni a megjelenítésben. Ezt a későbbiekben kiszínezem az egyik (vörös: *Reds*) színskálával.

```
symbol = QgsSymbol.defaultSymbol(layer.geometryType())
default_style = QgsStyle().defaultStyle()
```

```
ramp_name='Reds'
color_ramp = default_style.colorRamp(ramp_name)
```

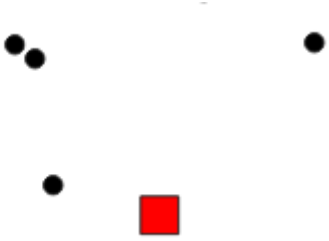
Végigmegyek a *unique_values* listán, amiben a különböző jogállások szerepelnek. Minden egyes jogálláshoz a *QgsRendererCategory()* függvénnyel létrehozok az érték alapján egy új szimbólumot és egy jelmagyarázatot. Végül a *QgsCategorizedSymbolRenderer(mezőnév, kategóriák listája)* kirajzoltatom a térképi vásznon (canvas) a jeleket. A szkriptet a nézet és rétegkezelő frissítésével zárom.

```
for unique_value in unique_values:
    category = QgsRendererCategory(unique_value, symbol, str(unique_value))
    categories.append(category)
renderer = QgsCategorizedSymbolRenderer(f,categories)

renderer.setClassAttribute(f)
renderer.updateColorRamp(color_ramp)

layer.setRenderer(renderer)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

12. feladat: Készítsünk kategorizált ábrázolást a település réteghez jogállás alapján!

12_categorized_styles2.py

Soroljuk csoportokba jogállás mező alapján a településeket. Két csoport legyen: község és nagyközség, valamint városok csoportja, ideértve a megyei jogú várost és megyeszékhelyet is. A kétféle településjel: egy fekete 2×2 mm-es kör és egy 4×4 mm-es vörös négyzet.



Miután hozzáfértünk az aktív réteghez, keressük meg, hogy a jogállás rétegen milyen értéket vehetnek fel az elemek (az előző példához hasonlóan). Ehhez a mező az indexét (hányadik oszlop a rétegen) kérdeztem le. Majd a réteg különböző értékeit a *uniqueValues()* metódussal szerzem meg, ezek az értékek egy listába kerülnek.

```
layer = iface.activeLayer()
f="jogallas"
fni = layer.dataProvider().fields().indexFromName(f)
unique_values = layer.uniqueValues(fni)
```

Ezután létrehozom mindkét szimbólumot a korábban tanultak alapján. A négyzetet a *QgsMarkerSymbol()* metódussal, a községjelet (mivel a rétegen a *default* marker alapbeállításként a pontmarker), ezért azt csak átszínezem.

```
categories=[]
color1=QColor('#000000')
```

```
symbol_v=QgsMarkerSymbol.createSimple ({'color':
'red','name':'square','size':4 })
symbol_k=QgsSymbol.defaultSymbol(layer.geometryType())
symbol_k.setColor(color1)
```

Ezután végignézem, hogy milyen értékeket vehetnek fel az egyes elemek. Ahol tartalmazza azt a szórészletet, hogy 'város' ott a város szimbólumot rendelem hozzá. Minden más község lesz.

Minden egyes jogálláshoz a *QgsRendererCategory()* függvénnyel létrehozok az érték alapján egy új szimbólumot és egy jelmagyarázatot. Végül a *QgsCategorizedSymbolRenderer(mezőnév, kategóriák listája)* kirajzoltatom a vásznon a jeleket. A szkriptet a nézet és rétegkezelő frissítésével zárom.

```
for unique_value in unique_values:
    if 'város' in unique_value:
        category = QgsRendererCategory(unique_value, symbol_v,
str(unique_value))
        categories.append(category)
    else:
        category = QgsRendererCategory(unique_value, symbol_k,
str(unique_value))
        categories.append(category)
renderer = QgsCategorizedSymbolRenderer(f,categories)

renderer.setClassAttribute(f)
layer.setRenderer(renderer)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

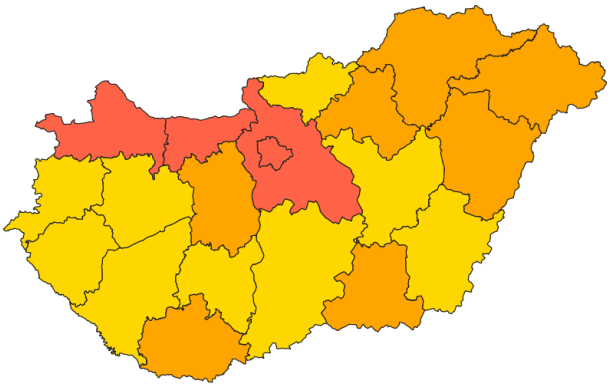
13. feladat: Készítsünk osztályozott ábrázolást a megye réteghez népsűrűség alapján!

13_graduated_styles.py

A feladatban használandó a megye réteg, ha nem szerepel benne még népsűrűség oszlop, készítsük el (elég egész számra kerekíteni az értékeket)!

Férjünk hozzá a behívott aktív réteghez. Osszuk 3 csoportba az adatokat, ezeket a következő színekkel jelenítsük meg: #FFD700, #FFA500, #FF6347

A három csoport legyen: 50-80, 81-100, 101 felett. (Nem ez a legjobb csoportba sorolási módszer, de most az egyszerűség kedvéért használjuk ezt).



Minden csoportnál írjuk be a min_val és max_val értéket, a színt (*QColor('#FFD700')*), majd a szimbólumot. A szimbólum a réteg *default* szimbóluma, ez felületrétegnél mindig egy random színű *Simple Fill*. Állítsuk be az új színt, majd a *QgsRendererRange* metódusban adjuk meg a legkisebb és legnagyobb határát a tartománynak, a szimbólumot és a csoport nevét. Végül pedig a csoportok listájához adjuk hozzá az utóbbi *range* változót.

```
layer = iface.activeLayer()

f="nepsur"
rangeList=[]
min_val=50
max_val=80
lab='Group 1'
color1=QColor('#ffee00')
symbol=QgsSymbol.defaultSymbol(layer.geometryType())
symbol.setColor(color1)
range1=QgsRendererRange(min_val, max_val, symbol, lab)
rangeList.append(range1)

min_val=81
max_val=100
lab='Group 2'
color2=QColor('#00eeff')
symbol=QgsSymbol.defaultSymbol(layer.geometryType())
symbol.setColor(color2)
range2=QgsRendererRange(min_val, max_val, symbol, lab)
rangeList.append(range2)

min_val=101
max_val=3500
lab='Group 3'
color3=QColor('#eeffff')
symbol=QgsSymbol.defaultSymbol(layer.geometryType())
symbol.setColor(color3)
range3=QgsRendererRange(min_val, max_val, symbol, lab)
rangeList.append(range3)
```

Ha megvannak a csoportok, rendereljük le őket az osztályozott renderelés metódussal. Két paramétert kell megadnunk, a mező nevét, ami alapján renderelünk (népsűrűség), valamint magukat a csoportokat, amelyeket szeretnénk létre hozni. Végül zárjuk a nézet, a réteg és a rétegkezelő frissítésével a sort.

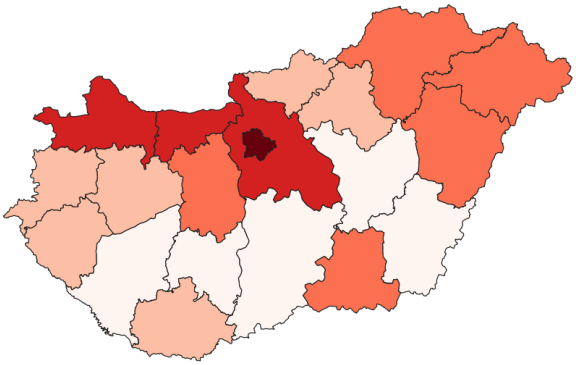
```
groupRenderer = QgsGraduatedSymbolRenderer(f,rangeList)
layer.setRenderer(groupRenderer)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

14. feladat: Készítsünk osztályozott ábrázolást a megye réteghez népsűrűség alapján! Az osztályokba sorolás alapja legyen Jenk természetes törések (natural breaks) algoritmusa.

14_graduated_styles2.py

A feladatban használandó a megye réteg, ha nem szerepel benne még népsűrűség oszlop, készítsük el (elég egész számra kerekíteni az értékeket)!

Az aktív megye réteg népsűrűség mezőjében lévő adatokat fogjuk osztályokba sorolni (5 osztály), és hozzárendeljük a fehér-vörös színátmenetes színskálát (neve *Reds*). A színskálák neveit egyétként a *color ramp* lenyíló ablakában lehet kikeresni.



```
layer = iface.activeLayer()
f="nepsur"
ramp_name='Reds'
num_classes=5
classification_method = QgsClassificationJenks()
default_style = QgsStyle().defaultStyle()
color_ramp = default_style.colorRamp(ramp_name)
```

Állítsuk be az osztályozott renderelést (*QgsGraduatedSymbolRenderer()*). Adjuk meg a mezőt, osztályozás módszerét, majd frissítjük az osztályok számát a rétegen. Megadjuk a színskálát is. Végül renderelünk, és frissítjük a nézetet, a réteget és a rétegkezelőt.

```
renderer = QgsGraduatedSymbolRenderer()
renderer.setClassAttribute(f)
renderer.setClassificationMethod(classification_method)
renderer.updateClasses(layer, num_classes)
renderer.updateColorRamp(color_ramp)

layer.setRenderer(renderer)

layer.triggerRepaint()
layer_tree=iface.layerTreeView()
layer_tree.refreshLayerSymbology(layer.id())
```

Az automatikus osztályba sorolásnál jelenleg a következő módszerek elérhetők:

- `QgsClassificationJenks ()`
- `QgsClassificationQuantile()`
- `QgsClassificationLogarithmic()`
- `QgsClassificationPrettyBreaks()`
- `QgsClassificationEqualInterval()`
- `QgsClassificationFixedInterval()`
- `QgsClassificationStandardDeviation()`

Hozzáférés a réteg attribútum táblázatához, kijelölések

Az attribútum táblázat kezelése PyQGIS-ben

15. feladat: Férjünk hozzá az egyes elemekhez és vizsgáljuk meg, mik az attribútum adataik! Írassuk ki, milyen különböző értékeket vehet fel egy mező.

15_attribute_table.py

A feladathoz használhatjuk az *mo.gpkg* település rétegét. Férjünk hozzá az aktuális réteghez.

Írjuk ki egy megadott elem attribútumait. Az elemre hivatkozhatunk az *fid*-val. (*fid=feature id,* azaz egyedi azonosító a rétegen. A térinformatikai szoftverek ezt mindig elkészítik, 0-val indul). Az adott elemhez hozzáférni a rétegen keresztül lehet a *getFeature(index)* függvénnyel. A *feat.attributes()* kiírja egy elem összes attribútumát. Ha csak egy attribútumra van szüksége, akkor a mező sorszámát kell megadni az attribútum táblázatban (*feat.attributes()[1]),* vagy a *feat['mezőnév']*-vel írható ki ugyanez.

```
layer = iface.activeLayer()
feat=layer.getFeature(0)
print (feat.attributes())
print (feat.attributes()[1]) #településnevet írja ki
print (feat["nev"]) #szintén a településnevet írja ki
```

Abban az esetben, amikor a *getFeature(fid)* a *fid* nagyobb, mint az elemszám, üres listát kapok vissza.

```
#Invalid fid
feat=layer.getFeature(11111)
print (feat.attributes())
#--> [] nincs ilyen elem
```

Menjünk végig a réteg összes elemén, és írjuk ki mindegyik elem egy megadott mezőjében lévő attribútumot.

```
for feature in layer.getFeatures():
    print (feature.attributes()[1])
```

Már az előző feladatokban láthattuk, hogy ha a *layer.fields()* függvényt hívom meg, akkor egy listában visszakapom az attribútum táblázat összes mezőnevét. Ha ezen egy *for* ciklussal végigfutok, akkor le tudom kérdezni a réteg mezőjének indexét (0,1,2…) a *fields.indexOf(field.name())* módon. Az index-szel pedig ki tudom íratni az adott mezőben lévő attribútumokat (*feature.attributes()[1]*).

```
fields=layer.fields()
for field in fields:
    print (field,fields.indexOf(field.name()))
```

A fenti megoldás ciklus nélkül is megvalósítható, vagyis a mező indexe lekérdezhető így is.

Ehhez adjuk meg a mező nevét, majd a *layer.dataProvider().fields().indexFromName(f)* függvénnyel keressük is ki az indexet.

```
f="jogallas"
fni = layer.dataProvider().fields().indexFromName(f)
```

A mező indexét egyébként az elem felől is kikereshetjük a *fieldNameIndex('mezőnév')* függvénnyel.

```
feature.fieldNameIndex('nev')
```

Szintén a kategorizált ábrázolásnál már láthattuk, hogy lekérdezhető az egy mezőben lévő különböző adatok. Erre a *uniqueValues(mező indexe)* függvény használható az alábbi módon. Egy *unique_values* listába teszem bele az eredményt, majd kiíratom egy *for* ciklussal a lista elemeit.

```
f="jogallas"
fni = layer.dataProvider().fields().indexFromName(f)
unique_values = layer.uniqueValues(fni)
for unique_value in unique_values:
    print(unique_value)
```

A fenti eredmény egyszerűbben is kivitelezhető. A *layer.uniqueValues()* paraméteréül adjuk meg máshogy a mező indexét: erre használjuk a *layer.fields().indexOf('mezőnév')*.

```
unique_values = layer.uniqueValues(layer.fields().indexOf('jogallas'))
for unique_value in unique_values:
    print(unique_value)
```

Lekérdezések és kijelölések

16. feladat: Jelöljünk ki egy téglalap alakú területen belül lévő összes pontot.

16_selection1.py

Használjuk az *mo.gpkg* település rétegét. Hozzunk létre egy téglalapot a *QgsRectangle(ymin, xmin, ymax, xmax)* függvénnyel. Ezután használjuk a *QgsFeatureRequest*() metódust, hogy hozzáférhessünk a réteg elemeinek geometriájához és attribútum adataihoz. Ezen metódus *setFilterRect()* függvénye alkalmas arra, hogy egy téglalap alakú területen belül lévő elemeket lekérdezhessük.

Ezután írassuk ki a korábban tanult módon egy ciklussal a lekérdezett elemek adatait (csak a település nevét). Végül hajtsunk végre egy kijelölést is az elemeken.

```
rectangle=QgsRectangle(600000,200000,650000,250000)
request=QgsFeatureRequest().setFilterRect(rectangle)
for feat in layer.getFeatures(request):
    print(feat.attributes()[1])
layer.selectByRect(rectangle)
```

A kijelölés megszüntetése:

```
layer.removeSelection()
```

17. feladat: Kijelölés kifejezéssel

17_selection2.py

Használjuk az *mo.gpkg* település rétegét. Kérdezzük le a megyei jogú városokat. Az előbb használt *QgsFeatureRequest()* metódus paraméteréül adhatjuk a *QgsExpression()* függvényt, amellyel lehetőség van kifejezéseket írni. Majd a rétegen hívjuk meg a *getFeatures* függvényt a korábbi lekérdezés szerint. Végül írassuk ki az egyes elemek nevét.

***getFeauture()* vagy *getFeatures()*?**

Az első esetben a megadott elemet tudjuk lekérni (csak egyet az *fid* alapján, vagy iterálva egyesével a rétegen). A *getFeatures()* esetben pedig a kérésben (*request*) megadott elemeket tudjuk lekérni a rétegről, nyilván csak azok fognak visszatérni, amelyek megfelelnek a feltételnek. A második eredménye éppen ezért mindig egy lista.

```
expr="jogallas='megyei jogú város'"
request=QgsFeatureRequest(QgsExpression(expr))
features=layer.getFeatures(request)
for feature in features:
    print (feature['nev'])
```

Ha szeretnénk kifejezés alapján kijelölni az elemeket, akkor a magát a kifejezést *selectByExpression()* függvény paraméteréül kell adnunk.

```
layer.selectByExpression(expr)
```

Egyéb kijelölési parancsok:

A *layer.selectAll()* függvénnyel a réteg össze elemét ki tudjuk jelölni.

A *layer.selectedFeatureCount()* megmondja, hogy hány kijelölt elem van.

A *layer.selectByIds([0,1,2,3])* függvény paraméteréül egy listát adunk. A listában a kijelölni kívánt elemek *fid*-jei szerepeljenek. Hasonlóan működik a *layer.select([1,2,3,4,5,6])* függvény.

A kijelölést a *layer.invertSelection()* paranccsal fordíthatjuk meg.

Korábban már említettem, hogy a *layer.removeSelection()* minden kijelölést megszüntet a rétegen. A *layer.deselect([7,8])* -tel pedig bizonyos elemek (ide is *fid*-k listája szükséges) kijelölését szünteti meg.

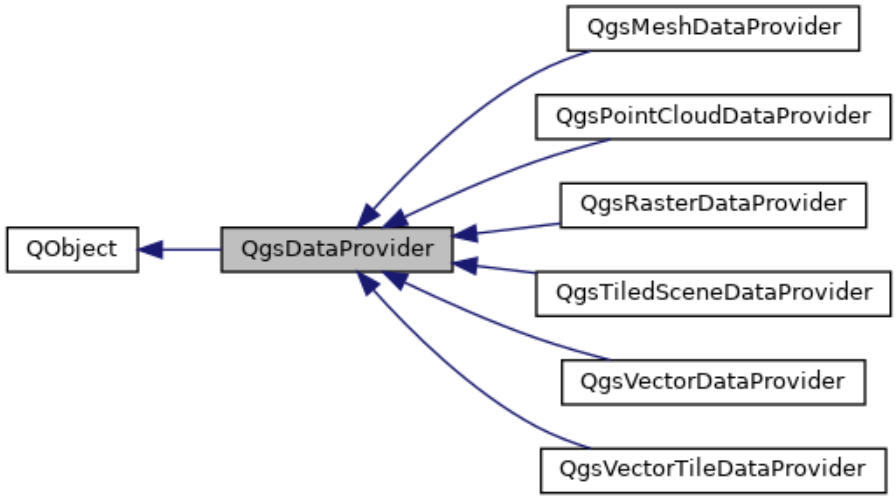
Attribútum adatok szerkesztése

Az attribútum adatokat kétféleképpen tudjuk szerkeszteni

- A data provideren keresztül: ennek az a hátránya, hogy nem lehet visszavonni.
- Az edit bufferen (szerkesztői pufferen) keresztül: akkor célszerű alkalmazni, ha a grafikus felületen (GUI) keresztül szerkesztjük az attribútum táblát. Ekkor lehetőség van visszavonni a módosításokat.

A dataProvider-ről

Az adatszolgáltatók (*dataProvider*) feladata a térbeli adatforrásból származó jellemző- és attribútuminformációk lekérdezése és írása (ahol ez támogatott). *A dataProvider* (adatszolgáltató) osztályban szerepelnek az adatok típusai szerinti szolgáltatók, pl. vektoros, raszteres, mesh stb.



18. feladat: Attribútum táblázat szerkesztése a data provideren keresztül

18_editing_attribute_table1.py

Változtassunk meg egy attribútumot a település réteg egyik elemén. A 3162-es elem duplumban tartalmazza Szentes települést (tulajdonképpen ennek egy településrésze, Belső Ecser). Írjuk át a településnevet!

Miután hozzáfértünk a réteghez, kapjuk el a réteg 3162. elemét. Hozzunk létre egy új szótárt (dictionary). A szótárban szerepeljen a mezőnév indexe, amelyet szeretnénk megváltoztatni, valamint a mező új értéke. A *feature.fieldNameIndex('nev')* függvénnyel kikereshetjük a mező indexét. A *changeAttributeValues(dictionary)* függvénnyel meg tudjuk változtatni az adott elem kiválasztott attribútumát.

```
layer=iface.activeLayer()
feature=layer.getFeature(3162)
new_name={feature.fieldNameIndex('nev'): 'Belső Ecser'}
layer.dataProvider().changeAttributeValues({feature.id():new_name})
```

Abban az esetben, ha nem csak egy, hanem több attribútumot is szeretnénk megváltoztatni, akkor a *field map*eket érdemes használni. A *field map*ek olyan szótárak, ahol a kulcs a mező neve, az érték pedig az indexe. Miután létrehoztunk az *attrs* szótárt, töltsük fel az új értékeket az kiválasztott elemhez adatokkal a *changeAttributeValues(dictionary)* függvénnyel. Ehhez korábban a *feature['mezőnév']= 'Új érték'*-ben már meg kellett adni az új adatokat.

```
layer=iface.activeLayer()
```

```
feature=layer.getFeature(3162)
feature['nev']='Belső Ecser'
feature['lakasok_2018']='50'
field_map=provider.fieldNameMap()
attrs={field_map[key]:feature[key] for key in field_map}
layer.dataProvider().changeAttributeValues({feature.id():attrs})
```

```
>>>field_map
{'fid': 0, 'jogallas': 3, 'ksh_kod': 2, 'lakasok_2018': 6, 'nepesseg_2018':
5, 'nev': 1, 'terulet_ha_2018': 4}
```

19. feladat: Attribútum táblázat szerkesztése az edit bufferen keresztül

19_editing_attribute_table2.py

Az edit buffert (szerkesztői puffer) akkor célszerű alkalmazni, ha a grafikus felületen (GUI) keresztül szerkesztjük az attribútum táblát. Ekkor lehetőség van visszavonni a módosításokat is. Ez talán egyszerűbben néz ki, mint a dataProvideren keresztüli szerkesztés. A feladat ugyanaz, mint az előbbiekben. A réteg szerkesztését a *startEditing()* függvénnyel indíthatjuk. Ebben a munkamenetben eszközölt változások nem lesznek tartósak, amíg meg nem hívjuk a *commitChanges()* függvényt. Ezután a *beginEditCommand()* függvénnyel lehet azt elérni, hogy a változások visszavonhatók legyenek, ezt pedig az *endEditCommand()* zárja le.

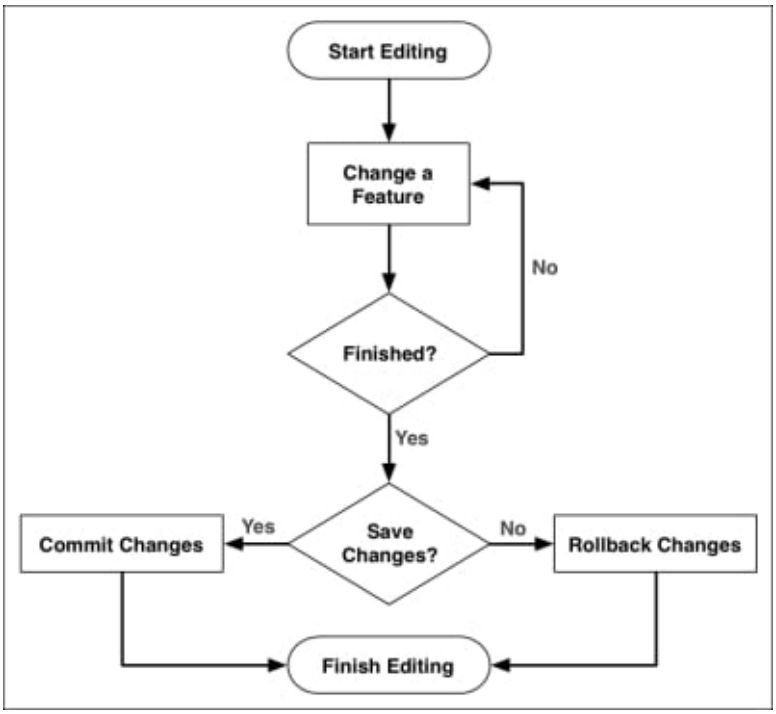
A *layer.changeAttributeValue()* függvényt pedig az előzőekhez hasonlóan alkalmazzuk. Megadjuk a módosítandó elem fid-ját, a mező id-jét, és az új értéket.

```
layer=iface.activeLayer()
layer.startEditing()
layer.beginEditCommand('Edit')
layer.changeAttributeValue(3162,1,'Belső Ecser')
layer.endEditCommand()
layer.commitChanges()
```

Az utóbbi így is módosítható, hasonlóan az előző feladathoz:

```
layer.changeAttributeValue(3162,layer.dataProvider().fieldNameIndex('nev'),
'Belső Ecser')
```

Az alábbi angol nyelvű ábra összefoglalja az edit buffer működését. Ezt használjuk tulajdonképpen, amikor a QGIS grafikus felületén szerkesztjük az adatokat.



20. feladat: Új mező létrehozása az attribútum táblázatban, valamint ennek feltöltése adatokkal

20_new_field.py

Hozzunk létre *mo.gpkg* település rétegének attribútum táblázatában két új mezőt X_eov és Y_eov néven. Ebbe írassuk be a település pontjelének X és Y koordinátáit.

Miután hozzáfértünk az aktív réteghez, először hozzunk létre két új oszlopot integer (egész szám) adattípussal. Ehhez először meg kell bizonyosodnunk arról, hogy a réteg rendelkezik-e ezzel a képességgel (*capabilities()*). Ha igen, akkor hívjuk meg a *layer.dataProvider().addAttributes()* függvényt. A *QgsField('mező neve', mező adattípusa')* paraméterekkel adható meg. A mező adattípusa lehet *QVariant.Int, QVariant.String, QVariant.Double*, ezek a leggyakrabban használt típusok. Végül az *updateFields()* mezővel véglegesítjük a változást.

Ezek után töltsük fel az új mezőket adattal. Nyissuk meg a réteg szerkesztését a korábban ismertetett módon a *startEditing()* és a *beginEditCommad()* függvényekkel (Az utóbbi függvény engedi a szerkesztések visszavonását is). Hozzunk létre két kifejezést, amely a réteg vetületében szerzi meg az X és Y koordinátákat (*QgsExpression('$x')*). A kifejezés kiértékelését a *QgsExpressionContext()* segíti. A kifejezéskörnyezetek azokat a paramétereket foglalják magukba, amelyekkel egy *QgsExpressiont()* ki kell értékelni. Az *appendScopes()* a hatókörök listáját csatolja a kontextus végéhez. Ez a hatókör felülírja a kontextuson belül a meglévő hatókörök által biztosított összes megfelelő változót vagy függvényt. A hatókörök tulajdonjoga átkerül a veremre, innen történik az adatok kiíratása. (Hogy mi az a verem – angolul stack – arról itt lehet olvasni https://hu.wikipedia.org/wiki/Verem_(adatszerkezet). Röviden, ha sok adat van a memória kezelését, és annak felszabadítását oldja meg a veremkiíratás).

```
layer=iface.activeLayer()
caps = layer.dataProvider().capabilities()
if caps & QgsVectorDataProvider.AddAttributes:
```

```
    res = layer.dataProvider().addAttributes([QgsField('X_eov',
QVariant.Int),
    QgsField('Y_eov', QVariant.Int)])
    layer.updateFields()
layer.startEditing()
layer.beginEditCommand('Edit')

expr_x=QgsExpression('$x')
expr_y=QgsExpression('$y')
context = QgsExpressionContext()
context.appendScopes(QgsExpressionContextUtils.globalProjectLayerScopes(lay
er))
provider=layer.dataProvider()
```

Ezután menjünk végig az egyes elemeken és értékeljük ki mindenhol a kifejezést (koordináta kiíratását). A *for* ciklusban a *feature* nevű változóba kerülnek bele az egyes elemek (geometria és leíró adatok). A *setFeature()* függvénnyel helyezzük el a kontextusban az elemet, hogy aztán a tulajdonságait (új mezők) frissíteni tudjuk. Adjuk meg a mezők értékét a kifejezések kiértékelésével.

A korábbiakhoz hasonlóan hozzunk létre egy *field map*et (speciális szótár), amelyben a kulcs a mező neve, az indexe pedig az értéke. Ez alapján már a *changeAttributeValues()* függvényben ténylegesen feltölthetjük a mezőket az új értékeikkel. Végül zárjuk le a réteg szerkesztését az *endEditCommand()* és a *commitChanges()* függvényekkel.

```
for feature in layer.getFeatures():
    context.setFeature(feature)
    feature['X_eov']=expr_x.evaluate(context)
    feature['Y_eov']=expr_y.evaluate(context)
    field_map=provider.fieldNameMap()
    attrs={field_map[key]:feature[key] for key in field_map}
    layer.dataProvider().changeAttributeValues({feature.id():attrs})
layer.endEditCommand()
layer.commitChanges()
```

21. feladat: Attribútum tábla egy oszlopának törlése

21_delete_field.py

Töröljük ki az *mo.gpkg* település táblájából a *lakasok_2018* oszlopot.

Miután hozzáfértünk az aktív réteghez, keressük meg az *indexFromName('mezőnév')* függvénnyel annak a mezőnek az indexét, amelyet törölni szeretnénk. Ezután hívjuk meg a *deleteAttributes()* függvényt, majd frissítsük az attribútum táblát az *updateField()* függvénnyel.

```
layer=iface.activeLayer()
field = layer.fields().indexFromName('lakasok_2018')
layer.dataProvider().deleteAttributes([field])
layer.updateFields()
```

Műveletek az elemek geometriájával: létrehozás, módosítás, elem törlése

22. feladat: Olvassuk be a geometriákat a teljes rétegről, különböző geometriai típusok esetén!

22_reading_geometries.py

Mivel nem biztos, hogy minden típusú réteg rendelkezésre áll az *mo.gpkg*-ban, ezért ahol szükséges, készítsünk egy új réteget a megfelelő geometriai típussal (Point, Linestring, Polygon, MultiPoint, MultiLinestring, MultiPolygon). Vegyünk fel rá néhány elemet! A feladatban kiolvassuk és kiíratjuk minden elem X és Y koordinátáját.

Hozzáférünk az aktív réteghez, elkapjuk az rétegen az egyes elemeket (*features*) a már ismert módon. Mindegyik feladat így kezdődik.

A geometriák megszerzéséhez a *geometry()* függvényt kell meghívni a *feature*-ön. Attól függően, hogy mi a geometria típusa, használjuk az *asPoint(), asMultiPoint(), asPolyline(), asMultiPolyline(), asPolygon(),* és az *asMultiPolygon()* függvényeket. Ezek WKB formátumú geometriává alakítják át a geometriát, amelyekkel már könnyen tudunk dolgozni.

```
layer=iface.activeLayer()
features=layer.getFeatures()
```

Pontrétegnél a legegyszerűbb a helyzet: egy ciklus szükséges ahhoz, hogy végignézzük a réteg elemeit. Majd az *x()* és *y()* függvényekkel íratjuk ki a koordinátákat.

```
for feat in features:
    geom=feat.geometry().asPoint()
    print (geom.x())
    print (geom.y())
```

A pontcsoportoknál egy további ciklus szükséges, hogy a csoport tagjain is végigmenjünk az elemek mellett.

```
for feat in features:
    p=1
    for part in feat.geometry().asMultiPoint():
        print ("part"+str(p))
        print(part.x())
        print(part.y())
        p=p+1
```

Az egyszerű vonalakat két ciklussal oldjuk meg: egyikkel az elemeken, a másikkal a vonal csomópontjain futok végig.

```
for feat in features:
    for point in feat.geometry().asPolyline():
        print(point.x())
        print(point.y())
```

A vonalcsoportoknál három ciklus szükséges: az elsővel a réteg elemeit vesszük sorra. A másodikkal az egyes elemek részeit, a harmadikkal pedig a rész csomópontjait íratjuk ki.

```
for feat in features:
    p=1
    for part in feat.geometry().asMultiPolyline():
        print("part"+str(p))
        for point in part:
            print(point.x(), point.y())
```

```
        p=p+1
```

A felületeknél szintén három ciklus szükséges. Az elsővel a réteg elemein megyek végig. A másodikkal a gyűrűket veszem számba. A poligonnak az első gyűrűje a külső gyűrű, a második és további gyűrűi a belső gyűrűk, vagyis lyukak. A harmadik ciklus ezen gyűrűk csomópontjainak beolvasásásához szükséges.

```
for feat in features:
    r=1
    for ring in feat.geometry().asPolygon():
        print("ring"+str(r))
        for point in ring:
            print(point.x(), point.y())
        r=r+1
```

Végül a poligoncsoport a legösszetettebb mind közül, itt négy ciklus lesz. Az elsővel a réteg elemeit olvasom be, a másodikkal megnézem, hogy egy csoport hány részből áll, és az egyes részeken megyek végig. Majd az egyes részek gyűrűit vizsgálom meg (az első gyűrű a külső gyűrű, a továbbiak a belső gyűrűk, vagyis a lyukak). Végül ezek csomópontjait íratom ki.

```
for feat in features:
    p=1
    for part in feat.geometry().asMultiPolygon():
        r=1
        print("part"+str(p))
        for ring in part:
            print("ring"+str(r))
            for point in ring:
                print(point.x(), point.y())
            r=r+1
        p=p+1
```

A feladat megoldásában a részeket *p*-vel a gyűrűket pedig *r*-rel jeleztem a könnyebb érthetőség végett.

23. feladat: Módosítsuk egy meglévő elem geometriáját a dataProvideren keresztül

23_modify_geometry1.py

Ehhez a feladathoz létrehoztam egy új réteget (a formátum mindegy). Valahol az országban vegyünk fel egy pontot. A pont új koordinátája legyen 650 000 és 200 000 (EOV-ben). A *dataProvider*en keresztüli módosítás folyamata lényegében megegyezik az attribútum táblázat módosításával (a megfelelő függvényeket kell csak kicserélnünk).

Miután hozzáférek az aktív réteghez, megkeresem a módosítandó elemet a *layer.getFeature(fid)* függvénnyel. Megvizsgálom, hogy létező elemről van- e szó (*isValid()*)?

A *QgsPoint(X,Y)* függvény arra való, hogy kétdimenziós pont elemeket hozzunk létre, amelyeknél a QGIS minimális memóriafelhasználással dolgozik. Ha egy kétdimenziós pont elemből pontszerű geometriát akarunk létrehozni a *QgsGeometry* osztály *fromPointXY()* függvényével tudom megtenni. Ezek után már csak annyi a feladatunk, hogy egy *geometry map*et készítünk (lényegében ugyanaz, mint a *field map*). Majd a *changeGeometryValues(geometry_map)* függvénnyel lecserélem a geometriát, és újrarajzoltatom a réteget (hogy térképi vászon is frissüljön).

```
 layer=iface.activeLayer()
feature=layer.getFeature(1)
if feature.isValid():
```

```
    geometry=QgsGeometry.fromPointXY(QgsPointXY(650000,200000))
    geometry_map={feature.id():geometry}
    layer.dataProvider().changeGeometryValues(geometry_map)
    layer.triggerRepaint()
```

24. feladat: Módosítsuk egy meglévő elem geometriáját az edit bufferen keresztül

24_modify_geometry2.py

Ehhez a feladathoz létrehoztam egy új réteget (a formátum mindegy). Valahol az országban vegyünk fel egy pontot. A pont új koordinátája legyen 600 000 és 200 000 (EOV-ben). Az edit bufferen keresztüli módosítás folyamata lényegében megegyezik az attribútum táblázat módosításával (a megfelelő függvényeket kell csak kicserélnünk).

A réteg szerkesztését a *startEditing()* függvénnyel indíthatjuk. Ebben a munkamenetben eszközölt változások nem lesznek tartósak, amíg meg nem hívjuk a *commitChanges()* függvényt. Ezután a *beginEditCommand()* függvénnyel lehet azt elérni, hogy a változások visszavonhatók legyenek, ezt pedig az *endEditCommand()* zárja le.

A *QgsPoint(X,Y)* függvény arra való, hogy kétdimenziós pont elemeket hozzunk létre, amelyeknél a QGIS minimális memóriafelhasználással dolgozik. Ha egy kétdimenziós pont elemből pontszerű geometriát akarunk létrehozni a *QgsGeometry* osztály *fromPointXY()* függvényével tudom megtenni. Végül pedig a geometria megváltoztatását a *layer.changeGeometry(fid, új geometria)* paranccsal lehet megtenni.

```
layer=iface.activeLayer()
layer.startEditing()
layer.beginEditCommand('Edit')
geometry=QgsGeometry.fromPointXY(QgsPointXY(600000,200000))
layer.changeGeometry(1,geometry)
layer.endEditCommand()
layer.commitChanges()
```

25. feladat: Az elem törlése a rétegről

25_delete_feature.py

Ehhez a feladathoz létrehoztam egy új réteget (a formátum mindegy). Valahol az országban vegyünk fel egy pontokat. Elemet törölni talán a legegyszerűbben az edit bufferen keresztül lehet. Hasonlóan az előzőekhez aktiváljuk a szerkesztést, majd *layer.getFeature(fid)* elkapjuk az elemet, és a *layer.deleteFeature(feature.id())*-val töröljük. Végül lezárjuk a szerkesztést a tanultak szerint.

```
layer=iface.activeLayer()
layer.startEditing()
layer.beginEditCommand('Edit')
feature=layer.getFeature(1)
layer.deleteFeature(feature.id())
layer.endEditCommand()
layer.commitChanges()
```

A fenti szerkesztés lerövidíthető is: a szerkesztést elindító és lezáró függvények elhagyhatók, helyette *a with edit(réteg neve)* használandó. Az *edit* osztály a *qgis.core* osztály része.

```
with edit(layer):
```

```
    feature=layer.getFeature(3)
    layer.deleteFeature(feature.id())
```

A *with* utasítás az általánosan használt *try/finally* hibakezelő utasításokat helyettesíti. A *with* utasítás használatának gyakori példája még egy fájl megnyitása.

26. feladat: Új pont létrehozása

26_create_new_point.py

Készítsünk egy új, üres réteget pont geometriával és egy mezővel az attribútum táblában. Hozzunk létre egy új pontot a rétegen.

A *capability* azt adja meg, hogy az adott rétegen milyen műveleteket tudunk végrehajtani. Ez a réteg típusától függően (pl. vektor, raszter stb.) eltérhet. A vektoros rétegen végrehajtható műveleteket ez a weboldal tartalmazza:

https://qgis.org/pyqgis/3.34/core/QgsVectorDataProvider.html#qgis.core.QgsVectorDataProvider

Ezek után hozzáadunk egy elemet a réteghez, ehhez meg kell nézni, hogy a *dataProvider*en ezt végre lehet-e hajtani, vagyis igaz-e az elágazás ezen része (*QgsVectorDataProvider.AddFeatures*). Az elemet a *QgsFeature()* függvénnyel adjuk hozzá, amelynek aztán a *setAttributes()* és a *setGeometry()* metódusával állítható be az attribútum táblázat tartalma, valamint a geometria. *QgsPointXY(X,Y)* függvény egy kétdimenziós pontot határoz meg. Ez még önmagában nem egy geometria, hanem egy koordinátapár (pl. az egérpozíció aktuális helyzete a képernyőn). Ebből a *QgsGeometry.fromPointXY()* függvény készíti el a tényleges geometriát. Végül adjuk hozzá az elkészült elemet a réteghez az *addFeatures()* függvénnyel.

```
layer=iface.activeLayer()
caps = layer.dataProvider().capabilities()
if caps & QgsVectorDataProvider.AddFeatures:
    feature = QgsFeature(layer.fields())
    feature.setAttributes([ 'egyes',1])
    feature.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(650257,
200706)))
    layer.dataProvider().addFeatures([feature])
layer.triggerRepaint()
```

27. feladat: Új vonal létrehozása

27_create_new_line.py

Készítsünk egy új üres réteget vonalas geometriával és egy mezővel az attribútum táblában. Hozzunk létre egy új vonalat a rétegen.

A feladat megoldása megegyezik az előzőével. Az egyetlen különbség, a vonal elkészítéséhez a *fromPolylineXY()* függvényt használjuk, melyben a pontokat egy listaként adhatjuk meg. A csomópontokat pedig az előzőekhez hasonlóan koordinátapárokként definiáljuk.

```
layer=iface.activeLayer()
caps = layer.dataProvider().capabilities()
if caps & QgsVectorDataProvider.AddFeatures:
    feature = QgsFeature(layer.fields())
    feature.setAttributes(['mezo'])
    feature.setGeometry(QgsGeometry.fromPolylineXY([QgsPointXY(650000,
200000),QgsPointXY(655000, 201000)]))
```

```
    layer.dataProvider().addFeatures([feature])
layer.triggerRepaint()
```

28. feladat: Új felület létrehozása

28_create_new_polygon.py

Készítsünk egy új, üres réteget felület geometriával és egy mezővel az attribútum táblában. Hozzunk létre egy új lyukas felületet a rétegen.

A feladat megoldása megegyezik az előzőével. Az egyetlen különbség, a felület elkészítéséhez a *fromPolygonXY()* függvényt használjuk, melyben a pontokat egy listaként adhatjuk meg. A csomópontokat pedig az előzőekhez hasonlóan koordinátapárokként definiáljuk. A poligonok szögletes zárójelbe tesszük, a külső foglalja magába a poligont, a belsők közül az első a poligon külső gyűrűjét, a továbbiak a belső gyűrűket (lyukakat) adják meg *[[külső gyűrű],[belső gyűrű]]*. Figyeljünk arra, hogy a poligon első csomópontja megismétlődik az utolsóként.

```
layer=iface.activeLayer()
caps = layer.dataProvider().capabilities()
if caps & QgsVectorDataProvider.AddFeatures:
    feature = QgsFeature(layer.fields())
    feature.setAttributes(['mezo'])
    feature.setGeometry(QgsGeometry.fromPolygonXY([[QgsPointXY(650000,
200000),QgsPointXY(650000, 201000),QgsPointXY(655000,
201000),QgsPointXY(655000, 200000),QgsPointXY(650000,
200000)],[QgsPointXY(651000, 200500),QgsPointXY(652000,
200400),QgsPointXY(651400, 200500),QgsPointXY(651000, 200500)]]))
    layer.dataProvider().addFeatures([feature])
layer.triggerRepaint()
```

29. feladat: Új csoportgeometria létrehozása

29_create_new_multigeometry.py

Készítsünk egy új, üres réteget felület geometriával és egy mezővel az attribútum táblában. Hozzunk létre egy felületcsoportot a rétegen.

A feladat megoldása megegyezik az előzőével. Az egyetlen különbség, a felületcsoport elkészítéséhez a *fromMultiPolygonXY()* függvényt használjuk, melyben a pontokat egy listaként adhatjuk meg. A csomópontokat pedig az előzőekhez hasonlóan koordinátapárokként definiáljuk. A poligonok szögletes zárójelbe tesszük, a külső foglalja magába a poligoncsoportot, majd a következők a poligont: ezek közül első a poligon külső gyűrűjét, a továbbiak a belső gyűrűket (lyukakat) adják meg: *[[[egyik poligon],[belső gyűrűje]],[[másik poligon]]]*. Figyeljünk arra, hogy a poligon első csomópontja megismétlődik az utolsóként.

```
layer=iface.activeLayer()
caps = layer.dataProvider().capabilities()
if caps & QgsVectorDataProvider.AddFeatures:
    feature = QgsFeature(layer.fields())
    feature.setAttributes(['mezo'])
    feature.setGeometry( QgsGeometry.fromMultiPolygonXY([[[
QgsPointXY(650000, 200000), QgsPointXY(650000, 201000), QgsPointXY(655000,
201000), QgsPointXY(655000, 200000),QgsPointXY(650000, 200000)],
[QgsPointXY(651000, 200500),QgsPointXY(652000, 200400),QgsPointXY(651400,
200500), QgsPointXY(651000, 200500)]], [[QgsPointXY(645000, 201000),
```

```
QgsPointXY(645000, 200000),QgsPointXY(642000, 200000),QgsPointXY(645000,
201000)]]]))
    layer.dataProvider().addFeatures([feature])
layer.triggerRepaint()
```

Ha vonalcsoportokat akarunk létrehozni, akkor a fromMultiPolylineXY([[első vonalszakasz pontjai],[második vonalszakasz pontjai]]) függvénnyel tehetjük meg.

```
#feature.setGeometry(QgsGeometry.fromMultiPolylineXY([[QgsPointXY(650000,
200000),QgsPointXY(655000, 201000)],[QgsPointXY(650000,
200000),QgsPointXY(650000, 205000)]]))
```

Ha pontcsoport létrehozása a cél, akkor *fromMultiPointXY([első pont, második pont])* függvénnyel oldjuk meg.

```
    feature.setGeometry(QgsGeometry.fromMultiPointXY([QgsPointXY(650257,
200706),QgsPointXY(650267, 200706)]))
```

Megemlíteném, ha nem elegendő a kétdimenziós geometria, akkor a függvények nevéből távolítsuk el az XY névtagot valahogy így: *fromPoint()*. Így lehetőség van Z vagy M koordináta megadására is.

Új réteg létrehozása

30. feladat: Új vektoros réteg létrehozása

30_create_new_vector_layer.py

Hozzunk létre egy új ideiglenes másnéven *temporary* réteget (*memory layer*). A réteg pontszerű adatokat tartalmaz, továbbá adjunk hozzá három mezőt is az attribútum táblába.

A *QgsVectorLayer("Point", "temporary_points", "memory")* függvénnyel adjuk meg a réteg típusát, nevét és az adatszolgáltató nevét. A *dataProvider.addAttributes* függvényével, miután elkezdjük szerkeszteni a réteget, új mezők adhatók hozzá. A *QgsField("mezo0", QVariant.String)* megadja a mező nevét és adattípusát.

Állandósítsuk a változásokat a rétegen a *commitChanges()* függvénnyel, majd a projekthez adjuk hozzá az új réteget (*QgsProject.instance().addMapLayer(layer)*).

```
layer = QgsVectorLayer("Point", "temporary_points", "memory")
provider = layer.dataProvider()

layer.startEditing()
provider.addAttributes( [ QgsField("mezo0",
QVariant.String),QgsField("mezo1",  QVariant.Int),QgsField("mezo2",
QVariant.Double) ] )
layer.commitChanges()
QgsProject.instance().addMapLayer(layer)
```

Hogyha a réteg vetületét szeretnénk megadni, akkor a réteg geometriai típusa után írjuk be sztringbe a következőképpen *?crs=epsg:EPSGAZONOSÍTÓ*. A geometriai típus lehet Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, stb.

```
layer = QgsVectorLayer("Point?crs=epsg:23700", "temporary_points",
"memory")
```

31. feladat: A vektoros réteg fájlba mentése

31_create_new_vector_file.py

Hozzunk létre egy új, üres vektoros réteget, majd tegyünk fel rá egy pontot. A fájlnévnél adjuk meg a teljes elérési utat! Hozzuk létre a mezők listát a korábbiakhoz hasonlóan. Hívjuk meg a *QgsVectorFileWriter()* függvényt. A függvényt a következőképpen paraméterezzük: elérési út és fájlnév, karakterkódolás, mezők, geometria típusa, vetület, fájl típusa.

A *QgsWkbTypes* sokféle lehet, itt a leggyakrabban használt bináris geometriák elnevezése:

Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon

Továbbiak: https://qgis.org/pyqgis/3.0/core/Wkb/QgsWkbTypes.html?highlight=qgswkbtypes

A QgsCoordinateReferenceSystem('EPSG:epsgazonosítószám')-vel adható meg a vetület.

Ezek után létrehozunk egy új *feature*-t, melynek megadjuk a geometriáját és attribútum adatait. Végül adjuk hozzá a réteghez az elemet az *addFeature()* függvénnyel. Ha befejeztük a fájl szerkesztését, mindenképpen zárjuk le a *del(writer)*-rel. Végül adjuk hozzá a létrehozott új réteget a projekthez.

```
out_name=" data/new_file_name.shp"
fields=QgsFields()
fields.append(QgsField("mezo0", QVariant.String))
```

```
fields.append(QgsField("mezo1", QVariant.Int))
fields.append(QgsField("mezo2", QVariant.Double))

writer=QgsVectorFileWriter(out_name,'UTF-
8',fields,QgsWkbTypes.Point,QgsCoordinateReferenceSystem('EPSG:23700'),'ESR
I Shapefile')

feature=QgsFeature()
feature.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(650000,200000)))
feature.setAttributes(['helló',1,1.5])
writer.addFeature(feature)
del(writer)
iface.addVectorLayer(out_name,'','ogr')
```

Ha nem Shapefilet, hanem Geopackaget szeretnénk készíteni a következő módosítások szükségesek. Először is az *out_name* változóban adjuk meg a GPKG nevét, pl. *new_file.gpkg*

Majd a fájlba írás előtt adjuk meg a réteg nevét, valamint a GPKG lesz a *QgsVectorFileWriter* függvényben a fájlformátum. Vigyázzunk, mert ezzel a módszerrel felülírjuk a GPKG-t, nem új réteget adunk hozzá!

```
options = QgsVectorFileWriter.SaveVectorOptions()
options.layerName = 'myLayer2'
writer=QgsVectorFileWriter(out_name,'UTF-
8',fields,QgsWkbTypes.Point,QgsCoordinateReferenceSystem('EPSG:23700'),'GPK
G',)
```

Vetületek kezelése

32. feladat: Vetületek kezelése: a projekt vetületének beállítása

32_working_with_projections.py

Állítsuk a réteg vetületét EPSG 23700-ról 3857-re.

Először meg kell adni az új vetületet a következőképpen: Q*gsCoordinateReferenceSystem('EPSG:avetületepsgazonosítója')*. Majd hozzáférünk a projekthez, és beállítjuk a *setCrs()* függvénnyel az új vetületet.

```
layer=iface.activeLayer()
newcrs=QgsCoordinateReferenceSystem("EPSG:3857")
QgsProject.instance().setCrs(newcrs)
```

33. feladat: Vetületek kezelése: a réteg vetületének beállítása

33_layer_projection.py

Az aktív réteget mentsük el más vetületben, ehhez használjuk a megye réteget. Tehát a kiindulási vetület a 23700 (EOV) és a cél pedig a 3857 (Web Mercator/szögtartó hengervetület WGS84-es alapfelületen).

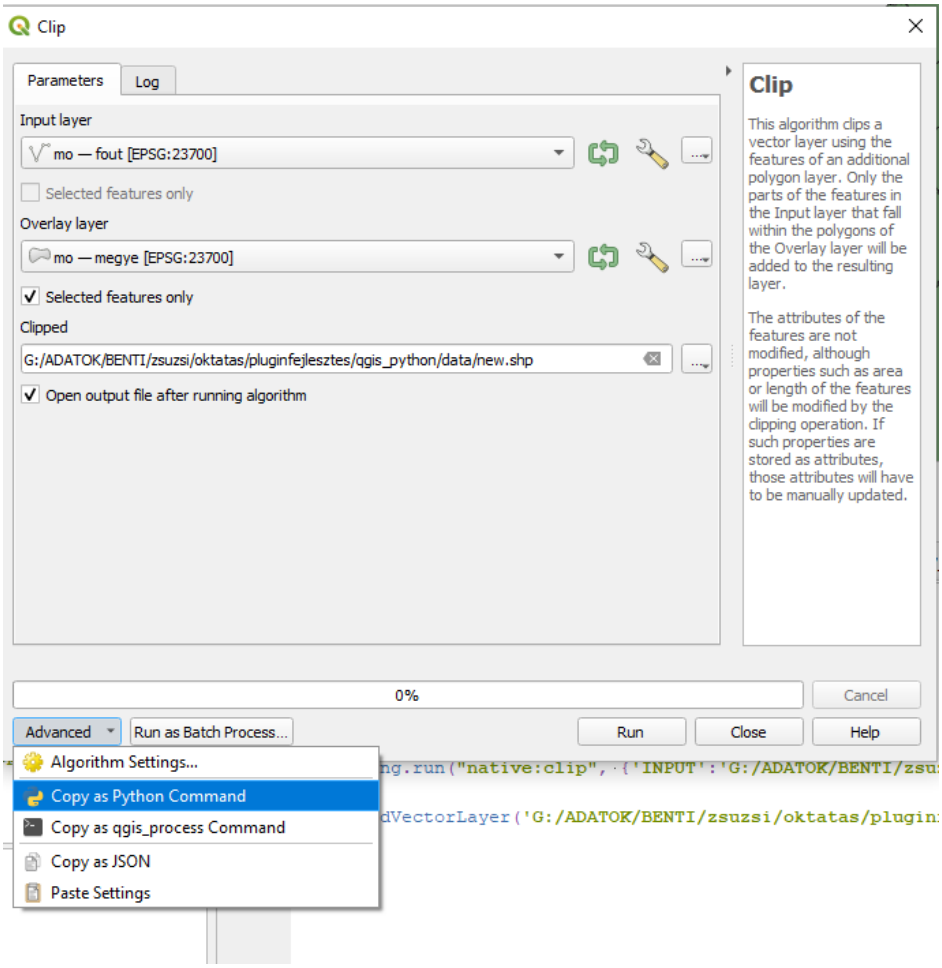
Miután megkerestük az aktív réteget, adjuk meg az új fájl nevét (ez legyen egy Shapefile). A *crs.authid()* függvénnyel tudjuk lekérni a réteg vetületének EPSG számát (*'EPSG:23700'*). Adjuk meg a Q*gsCoordinateReferenceSystem()* függvénnyel a kiindulási és célvetületet. Hozzunk létre egy új fájlt a *QgsVectorFileWriter()*-rel. Fussunk végig a megye réteg elemein a *getFeatures()* függvénnyel, majd a ciklus minden egyes lefutásában hozzunk létre egy új elemet. Hajtsuk végre egy vetületi transzformációt az elemen, majd ez a transzformált geometria legyen az új elem geometriája is *newfeat.setGeometry(geom).* Zárjuk le a fájlba írást, majd adjuk hozzá a projekthez.

```
layer=iface.activeLayer()
out_name=" C:/adatok/könyvtára/new_file_name.shp"
source_crs = QgsCoordinateReferenceSystem(layer.crs().authid())
target_crs = QgsCoordinateReferenceSystem("EPSG:3857")
fields=layer.fields()
writer=QgsVectorFileWriter(out_name,'UTF-
8',fields,QgsWkbTypes.Polygon,target_crs,'ESRI Shapefile')
for feature in layer.getFeatures():
    newfeat=QgsFeature()
    geom = feature.geometry()
    geom.transform(QgsCoordinateTransform(source_crs, target_crs,
QgsProject.instance()))
    newfeat.setGeometry(geom)
    writer.addFeature(newfeat)
del(writer)

iface.addVectorLayer(out_name,'','ogr')
```

A feldolgozó eszközök

A *QGIS Processing* menüpontjában található minden olyan eszköz, amely főleg a térbeli raszteres és vektoros adatok feldolgozásához kapcsolódik. Tulajdonképpen a *Raster* és *Vector* menüben található algoritmusok ennek egy szűk (belépő szintű) kivonata. A *Processing* menüben elérhető algoritmusokhoz azért hoztak létre Python kódot, hogy megkönnyítések a felhasználók számára ezen algoritmusok sorozatos használatát. Ezek ugyanarra a mintára készültek, csak az egyes algoritmusok paraméterezése tér el.

A feldolgozó eszközök használata úgy a legegyszerűbb, hogy kihasználjuk az épp aktuálisan megnyitott menüben – pl. *Clip* vagy bármelyik másik – azt a lehetőséget, hogy a menüben a beállításokat követően képesek vagyunk a Python kódot kinyerni. Ezt csak le kell futtatnunk a konzolban. A minta alapján már könnyen testre szabhatjuk a szkriptet a későbbi céljainknak megfelelően.



A feldolgozó eszközök a *Processing* modulban vannak, amely a *qgis.core* osztály részei, tehát külön importálni az újabb QGIS verziókban már nem szükséges. Ha mégis szeretnénk az *import processing* paranccsal tehetjük meg a kódunk fejlécében.

Lássuk először, hogy milyen elérhető algoritmusok vannak a *Processing* menüben:

```
for alg in QgsApplication.processingRegistry().algorithms():
        print(alg.id(), "->", alg.displayName())
```

Ha ezzel a kóddal kilistázom az algoritmusokat kb. 700 név fog érkezni jelen pillanatban, amelyet a jegyzet keretein belül áttekinteni lehetetlenség. Néhány fontosabbat válogattam ki, amellyel megérthető a feldolgozó algoritmusok használata.

Az algoritmusok paraméterezése két részből tevődik össze: először megadjuk a nevét, majd az algoritmus specifikus paramétereket. Az algoritmusok a következő kategóriákba vannak besorolva: 3d, gdal, grass7:(g, i, r, m, nviz, v), native, pdal, qgis – illetve, ha valakinek telepítve vannak további feldolgozó eszközök ennél jóval többféle lehet, pl. az Orfeo Toolbox (otb) további száz algoritmust ad hozzá.

```
processing.run(name_of_the_algorithm, parameters)
```

Az algoritmusok paraméterezésében úgy kaphatunk segítséget, ha lefuttatjuk a *help* funkciót az algoritmus nevén a következő kóddal. A paramétereket egy szótárban (*dictionary*-ben adjuk meg).

```
processing.algorithmHelp("native:symmetricaldifference")
```

Ezt kapjuk segítségképpen:

```
OUTPUT: Symmetrical difference
      Parameter type:   QgsProcessingParameterFeatureSink
      Accepted data types:
            - str: destination vector file, e.g. 'd:/test.shp'
            - str: 'memory:' to store result in temporary memory layer
            - str: using vector provider ID prefix and destination URI,
e.g. 'postgres:…' to store result in PostGIS table
            - QgsProcessingOutputLayerDefinition
            - QgsProperty
GRID_SIZE: Grid size
      Parameter type:   QgsProcessingParameterNumber
      Accepted data types:
            - int
            - float
            - QgsProperty
----------------
Outputs
----------------
OUTPUT:  <QgsProcessingOutputVectorLayer>
      Symmetrical difference
```

A következő hivatalos QGIS aloldalon kaphatunk segítséget a feldolgozó eszközökről:

https://docs.qgis.org/3.34/en/docs/user_manual/processing/console.html

A fentiek alapján készültek el a következő kódok némi magyarázattal.

34. feladat: Metszetszámítás (clip)

34_clip.py

Hívjuk be az *mo.gpkg*-ból a főutak és megye réteget. Jelöljük ki Fejér megye poligonját, és kérdezzük le a vele átfedő utakat. Ezeket írjuk ki egy *Shapefile*-ba.

A kijelölést a 17. feladatban megadott módon fogom végrehajtani, azzal a különbséggel, hogy most a megye nevét kérdezem le.

Ezután a *Processing* osztály *run* metódusát hívjuk meg, ezt kell a továbbiakban paraméterezni. A *native:'algoritmus neve'* paraméternél adható meg, hogy melyik algoritmust fogjuk futtatni, pl. *Clip*. A második paraméter pedig a meghívott algoritmus jellemzőit tartalmazza egy kapcsos zárójelben. A *Clip* esetén ezek az *INPUT, OVERLAY, selectedFeaturesOnly, featureLimit, geometryCheck, OUTPUT*.

Végül adjuk hozzá az interfészhez az elkészült réteget a már ismert módon.

```
expr="nev='Fejér'"
request=QgsFeatureRequest(QgsExpression(expr))
features=layer.getFeatures(request)
layer.selectByExpression(expr)

processing.run("native:clip", {'INPUT':'
C:/adatok/könyvtára/mo.gpkg|layername=fout',
'OVERLAY':QgsProcessingFeatureSourceDefinition('C:/adatok/könyvtára/
mo.gpkg|layername=megye',
selectedFeaturesOnly=True,
featureLimit=-1,
geometryCheck=QgsFeatureRequest.GeometryAbortOnInvalid),
'OUTPUT':'C:/adatok/könyvtára/new.shp'})

iface.addVectorLayer('C:/adatok/könyvtára/new.shp','','ogr')
```

35. feladat: Különbségszámítás (difference)

35_Difference.py

Hívjuk be az *mo.gpkg*-ból a főutak és megye réteget. Jelöljük ki Fejér megye poligonját, és vonjuk ki Fejér megye útjait a Magyarországból. Ezeket írjuk ki egy GPKG-ba.

A feladat az előzővel megegyezik, a *Difference* algoritmus paraméterezésében mutatkozik különbség.

INPUT, OVERLAY, selectedFeaturesOnly, featureLimit, geometryCheck, OUTPUT, GRID_SIZE.

```
processing.run("native:difference", {'INPUT':'
C:/adatok/könyvtára/mo.gpkg|layername=fout',
'OVERLAY':QgsProcessingFeatureSourceDefinition('
C:/adatok/könyvtára/mo.gpkg|layername=megye', selectedFeaturesOnly=True,
featureLimit=-1,
geometryCheck=QgsFeatureRequest.GeometryAbortOnInvalid),
'OUTPUT':'ogr:dbname=\' C:/adatok/könyvtára/new.gpkg\' table="roads"
(geom)',
'GRID_SIZE':None})
```

36. feladat: A réteg elemeinek összeolvasztása (dissolve)

36_Dissolve.py

Készítsünk egy országpoligont a megyékből. Ehhez a *Dissolve* funkciót használhatjuk. Bemeneti paraméterei a következők: *INPUT, FIELD, SEPARATE_DISJOINT, OUTPUT*. A *Field* paraméterben definiálható, hogy valamelyik közös attribútum alapján olvasztjuk össze a réteg elemeit, vagy enélkül. Itt az utóbbit használtam.

Végül adjuk hozzá az elkészült réteget.

```
processing.run("native:dissolve", {'INPUT':
C:/adatok/könyvtára/mo.gpkg|layername=megye',
'FIELD':[],
'SEPARATE_DISJOINT':False,
```

```
'OUTPUT': C:/adatok/könyvtára/megye_dissolved.shp'})

iface.addVectorLayer(C:/adatok/könyvtára/megye_dissolved.shp','','ogr')
```

37. feladat: Pufferzóna (Buffer)

37_Buffer.py

Az előbbi feladatban elkészült országhoz készítsünk pufferzónát 10 km szélességben. A pufferzóna elkészítésének több paramétere van, ezek: *INPUT, DISTANCE, SEGMENTS, END_CAP_STYLE, JOIN_STYLE, MITER_LIMIT, DISSOLVE, SEPARATE_DISJOINT, OUTPUT*. A *distance* a pufferzóna szélességét adja meg, a *segments* a részletességét (minél kisebb a szám, annál részletesebb), az *end_cap_style* a vonalvégek stílusát (0: *round*/lekerekített, 1: *flat*/egyenes, 2: *square*/egyenes, de túlnyúlik) a *join_style* pedig a vonal törésének jellegét (0: *round*/lekerekített, 1: *miter*/egyenes, egy pontszerű vég, 2: *beveled*/szögletesen lecsapott). A *miter_limit* az egyenes, pontszerű vég levágásának mértékét jelenti. A *dissolve* pedig az egymással átfedő pufferzónák összeolvasztásáról nyilatkozik.

```
processing.run("native:buffer", {'INPUT':
C:/adatok/könyvtára/megye_dissolved.shp',
'DISTANCE':10000,
'SEGMENTS':2,
'END_CAP_STYLE':0,
'JOIN_STYLE':0,
'MITER_LIMIT':2,
'DISSOLVE':True,
'SEPARATE_DISJOINT':False,
'OUTPUT': C:/adatok/könyvtára/megye_buffer1.shp'})

iface.addVectorLayer(C:/adatok/könyvtára/megye_buffer1.shp','','ogr')
```

Raszteres képek olvasása, pixelértékek megszerzése

38. feladat: Raszteres képek olvasása, eltávolítása

38_loading_raster_layer.py

A raszteres adatok beolvasása nagyon hasonlít a vektoros adatokéhoz, azzal a különbséggel, hogy a *QgsRasterLayer()* függvényt kell meghívnunk. A lentebb bemutatott két módszerrel bármilyen típusú raszteres képet behívhatunk, legyen az georeferált térkép, többcsatornás műholdfelvétel, vagy domborzatmodell. A közös bennük, hogy mindegyiket GeoTiff formátumban tárolom.

```
path= C:/adatok/könyvtára/srtm_orai.tif'
raster=QgsRasterLayer(path, 'srtm','gdal')
QgsProject.instance().addMapLayer(raster)
```

Másképpen, ha megvan az elérési út definíciója.

```
iface.addRasterLayer(path, "srtm")
```

A réteg eltávolítható a projektből a *removeMapLayer(layer.id())* függvénnyel.

```
layer=iface.activeLayer()
QgsProject.instance().removeMapLayer(layer.id())
```

39. feladat: Raszteres képek metaadatai

39_raster_infos.py

Kérjük le a következő információkat a raszteres képről.

Először a kép sorainak és oszlopainak számát.

```
layer=iface.activeLayer()
print(layer.width(), layer.height())
```

Majd a réteg kiterjedését. Alakítsuk szöveggé.

```
print(layer.extent())
print(layer.extent().toString())
```

Írjuk ki a kép típusát, ez lehet *0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband*

```
print(layer.rasterType())
```

Hány csatornás a kép?

```
print(layer.bandCount())
```

Kérjük le, hogy mely renderelőket használhatjuk a kép megjelenítéséhez. Ezek a következők lehetnek:

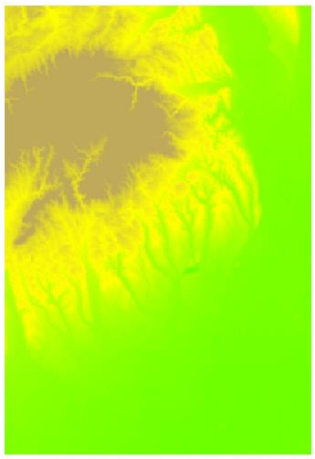
- `QgsHillshadeRenderer`
- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`
- `QgsRasterContourRenderer`
- `QgsSingleBandColorDataRenderer`
- `QgsSingleBandGrayRenderer`
- `QgsSingleBandPseudoColorRenderer`

```
print(layer.renderer().type())
```

40. feladat: A raszteres kép renderelése: domborzatmodell

40_raster_image_renderer1.py

Töltsünk be egy domborzatmodellt, például az *srtm_orai.tif*-et. Színezzük ki a domborzatot a következő magasságok alapján: 0 (zöld), 200 (sárga), 500 (barna). A színskála legyen színátmenetes (*Interpolated*).



Keressük meg az aktív réteget. A *QgsColorRampShader()* fogja kiszínezni a pixeleket az értékükhöz rendelt szín alapján. Ennek a domborzatmodelleknél a következő két fajtáját használjuk: *QgsColorRampShader.Interpolated, QgsColorRampShader.Discrete.* A *QgsColorRampShader().setColorRampType()* függvénye állítja be a színezés módszerét. A színskálát (*color ramp*) a *ColorRampItem(magasság, szín)*-mel tudjuk definiálni. A *setColorRampItemList()* állítja be az általunk listaként megadott szín és magasságskálát.

```
layer=iface.activeLayer()
colorshader= QgsColorRampShader()

colorshader.setColorRampType(QgsColorRampShader.Interpolated)

lst = [QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)),

      QgsColorRampShader.ColorRampItem(200, QColor(255,255,0)),
     QgsColorRampShader.ColorRampItem(500, QColor(190,170,90)) ]

colorshader.setColorRampItemList(lst)
```

A *QgsRasterShader()* fogja az egyes pixeleket a fentebb megadott tulajdonságok alapján kiszínezni/árnyalni. Végül rendereljük le a réteget az egysávos álszínes függvénnyel (*QgsSingleBandPseudoColorRenderer(a réteg dataProvidere, a band száma, QgsRasterShader))*. Végül pedig hívjuk meg a rétegen a renderelőt és frissítsük a nézetet.

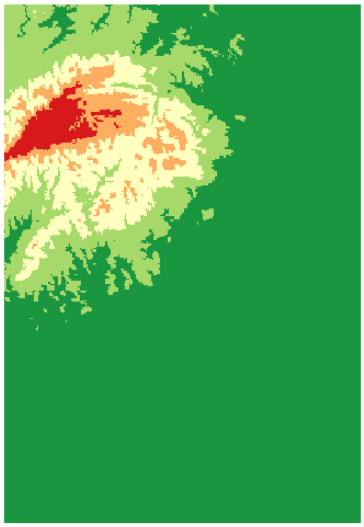
```
shader = QgsRasterShader()

shader.setRasterShaderFunction(colorshader)

renderer = QgsSingleBandPseudoColorRenderer(layer.dataProvider(), 1,
shader)
layer.setRenderer(renderer)
```

```
layer.triggerRepaint()
```

41. feladat: A raszteres kép renderelése meglévő színskála alapján

41_ raster_image_renderer2.py

Rendereljük le az egyik színskála szerint (RdYlGn) a *srtm_orai.tif* réteget. Vegyük ki a rétegből a legkisebb és legnagyobb értéket, ezek különbségét osszuk 5 egyenlő tartományra. A színskála megjelenítése legyen inverált és diszkrét.



Hozzáférünk az aktív réteghez.

```
layer = iface.activeLayer()
```

Kérjük le a réteg statisztikáját a *bandStatistics()* függvénnyel. Paraméterek a csatorna száma, a csatorna statisztikai adatai *(QgsRasterBandStats.All* – pl. legkisebb, legnagyobb érték, átlag, szórás stb.), a réteg térbeli kiterjedése, és a pixelek becsült darabszáma (ha ez 0, a teljes raszteres kép összes pixelét használja).

Írassuk ki a maximum és minimum értéket, valamint a kettő különbségét.

```
stats=layer.dataProvider().bandStatistics(1,QgsRasterBandStats.All,colourBa
ndLayer.extent(),0)
maxHeight = math.ceil(stats.maximumValue)
minHeight = math.floor(stats.minimumValue)
heightRange=maxHeight - minHeight
```

Hozzunk létre egy új stílust. Adjuk meg a színskálát a neve alapján, majd invertáljuk.

```
myStyle = QgsStyle().defaultStyle()
ramp_name='RdYlGn'
colorRamp = myStyle.colorRamp(ramp_name)
colorRamp.invert()
```

Hívjuk meg az egysávos álszínes renderelőt (*QgsSingleBandPseudoColorRenderer()*). Állítsuk be az osztályozás minimum és maximum értékét (*setClassificationMin()* és *setClassificationMax()*).

```
renderer = QgsSingleBandPseudoColorRenderer(layer.dataProvider(), 1)
renderer.setClassificationMin(minHeight)
```

```
renderer.setClassificationMax(maxHeight)
```

A *createShader()* függvény fogja létrehozni az árnyalást: beállítja a színskálát, az interpoláció típusát (diszkrét), az osztályozás módszerét (itt folytonos), és a csoportok számát. Ezután végre kell hajtani a renderelést és frissítsük a térképi vásznat.

```
renderer.createShader(colorRamp, QgsColorRampShader.Discrete,
QgsColorRampShader.Continuous, 5)
layer.setRenderer(renderer)
layer.triggerRepaint()
```

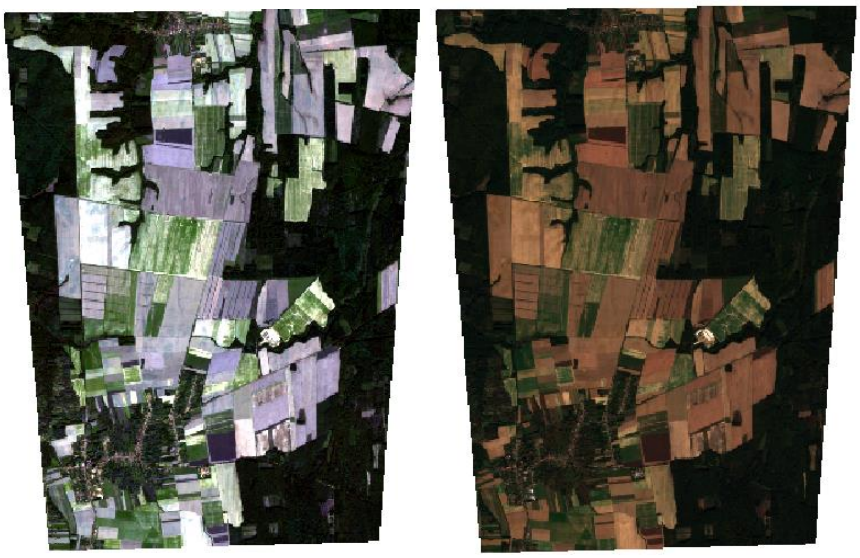
42. feladat: A raszteres kép renderelése: műholdkép

42_ raster_image_renderer3.py

Töltsük be *a sentinel_felvetel.tif* fájlt. A betöltött műholdkép többsávos, az 3-es a vörös, a 2-es a zöld, a 1-as pedig a kék csatornán érzékelt kép. Azonban a betöltéskor a 1-es és a 3-as csatorna felcserélődik. A feladat az, hogy állítsuk helyes sorrendbe a csatornákat.

Elkapjuk az aktív réteget. Majd hívjuk meg a rétegen *a setContrastEnhancement()* függvényt, amellyel a csatorna legkisebb és legnagyobb pixelértékére tudom feszíteni a kontrasztot: *QgsContrastEnhancement.StretchToMinimumMaximum.* Végül a réteg aktuális renderelőjén (ez ugye most a *MultiBand renderer*) állítsuk be a csatornák új kombinációját. A *setRedBand(csatorna száma)*, *setGreendBand(csatorna száma)* és a *setBlueBand(csatorna száma)* függvénnyel tudjuk megadni a csatornák új kombinációját. Végül frissítjük a réteget a térképi vásznon.

```
layer=iface.activeLayer()
layer.setContrastEnhancement(QgsContrastEnhancement.StretchToMinimumMaximum
)
layer.renderer().setRedBand(3)
layer.renderer().setGreenBand(2)
layer.renderer().setBlueBand(1)

layer.triggerRepaint()
```

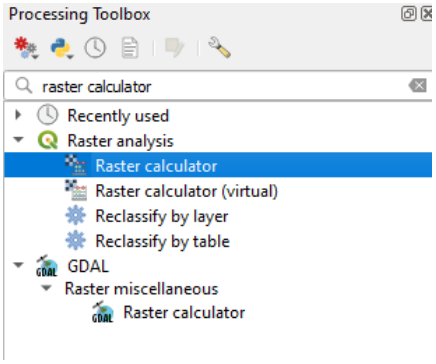


Raszteres képe írása, műveletek raszteres fájlokkal

43. feladat: A Raster Calculator menü használata

43_ raster_calculator.py

A raszteres képek feldolgozásához gyakran hívjuk segítségül a *Raster Calculator* menüt. Azonban, ha a *Raster* menü → *Raster Calculator* pontjából indítjuk, nem tudjuk letölteni a Python szkriptet, holott ez is a *Processing* plugin része. Ezért javaslom, hogy a *Processing Toolbox*ból indítsuk el a *Raster Calculator*t. A *Raster Calculator*, mint az alábbi ábrán is látszik, nem csak egy menü. Az első *Raster calculator ('native')* menüben a képhez, mint réteghez lehet hozzáférni, a csatornákat pedig a @csatorna száma módszerrel lehet megadni. Most használjuk a *GDAL Raster Calculator*át (utolsó látható modul az alábbi képen).



Innen a vektoros részben ismertetett módon nyerhető ki a szkript.

A feladat az lesz, hogy az számítsuk ki az NDVI indexet a *sentinel_felvetel.tif* műholdképhez. Ehhez adjuk meg, hogy mely sávok tartalmazzák a felvételen a közeli infravörös (Band 7) és vörös (Band 3) sávot.

$$\mathrm{NDVI} = \frac{(\mathrm{NIR} - \mathrm{Red})}{(\mathrm{NIR} + \mathrm{Red})}$$

Ezeket a paramétereket kellett felvinni a menüben. Az 'A' sáv legyen a Red, a 'B' sáv pedig a NIR. Így a képlet (B-A)/(B+A) lesz. A felhasználandó sávok mellett még az *Extent*-et és az *Output*-ot állítottam be.

```
processing.run("gdal:rastercalculator", {'INPUT_A':'
C:/adatok/könyvtára/sentinel_felvetel.tif',
'BAND_A':3,
'INPUT_B':' C:/adatok/könyvtára/sentinel_felvetel.tif',
'BAND_B':7,
'INPUT_C':None,'BAND_C':None,
'INPUT_D':None,'BAND_D':None,
'INPUT_E':None,'BAND_E':None,
'INPUT_F':None,'BAND_F':None,
'FORMULA':'(B-A)/(B+A)',
'NO_DATA':None,
'EXTENT_OPT':0,
'PROJWIN':'660467.187500000,666867.187500000,5135093.000000000,5143443.0000
00000 [EPSG:32633]',
'RTYPE':5,
'OPTIONS':'','EXTRA':'',
'OUTPUT':'C:/adatok/könyvtára/s.tif'})
```

Ha betöltjük az elkészült képet betöltjük a szürkeárnyalatos megjelenítésben, a következőt kapjuk:

Összetett feladatok vektoros és raszteres adatokkal

44. feladat: Vonalak simítása: helyettesítsük Bézier-görbével a szögletes vonalat. A Bézier-görbék legyenek folytonosak.

44_smoothing_with_bezier_curve.py

A feladatban adott egy Douglas–Peucker-algoritmussal egyszerűsített szintvonalrajz (*contour.shp*). Mivel a szintvonalak szögletes vonalak (polyline-ok), ezért ezek kinézetét szeretném simítani. A simításhoz Bézier-görbéket használok. Ezek a Bézier-görbék legyenek folytonosak. Mivel nincsenek meg a kontrollpontjaink, ezért azokat is ki kell számítanunk úgy, hogy a görbék folytonosak legyenek. A számításhoz szükség lesz az aktuális görbeszakasz töréspontjaira, valamint a szakaszt megelőző és a szakaszt követő első csomópontokra. A feladat végén olvashat a Bézier-görbékről szóló legfontosabb tudnivalókat. A következő egyenletekkel oldható meg:

Számítsuk ki *x* és *y* koordinátára külön-külön a *c*-t a szomszédos csomópontokból, ahol *t* a csomópontokat tartalmazó lista, *i* pedig az aktuális csomópont.

$$c_1 = \frac{t_{i-1} + t_i}{2}$$

$$c_2 = \frac{t_i + t_{i+1}}{2}$$

$$c_3 = \frac{t_{i+1} + t_{i+2}}{2}$$

Számítsuk ki a csomópontok közötti távolságokat is.

$$len_1 = \sqrt{(t_{xi} - t_{xi-1})^2 + (t_{yi} - t_{yi-1})^2}$$

$$len_2 = \sqrt{(t_{xi+1} - t_{xi1})^2 + (t_{yi+1} - t_{yi1})^2}$$

$$len_3 = \sqrt{(t_{xi+2} - t_{xi+1})^2 + (t_{yi+2} - t_{yi+1})^2}$$

majd ezekből számítsuk ki *k*-t.

$$k_1 = \frac{len_1}{(len_1 + len_2)}$$

$$k_2 = \frac{len_2}{(len_2 + len_3)}$$

Számítsuk ki *m*-et *x*-re és *y*-ra is.

$$m_1 = c_1 + k_1(c_2 - c_1)$$

$$m_2 = c_2 + k_2(c_3 - c_2)$$

Végül megkapjuk a két kontrollpontot *x*-re és *y*-ra is.

$$ctrl_1 = m_1 + sv(c_2 - m_1) + t_i - m_1$$

$$ctrl_2 = m_2 + sv(c_2 - m_2) + t_{i+1} - m_2$$

Végül a Bézier-görbe nyomvonalának kiszámítása a következőképpen történik (először helyettesítsünk be x majd y koordinátára), ahol **n** az esetünkben 1 és 10 között változó paraméter, ezzel lehet megadni, milyen sűrűn helyezkedjenek el csomópontok a görbe mentén.

$$B(n) = (1-n)^3 t_i + 3t(1-n)^2 ctrl_1 + 3n^2(1-n)ctrl_2 + n^3 t_{i+1}$$

A szintvonalak réteget olvassuk be, mint aktív réteg. Hozzunk létre egy ideiglenes (*temporary*) réteget EOV vetületben. Ezt az új réteget tegyük szerkeszthetővé.

```
layer=iface.activeLayer()
features=layer.getFeatures()
smooth_value=0.7
newlayer = QgsVectorLayer("LineString?crs=epsg:23700", "smoothed_lines",
"memory")
provider = layer.dataProvider()
caps = newlayer.dataProvider().capabilities()
newlayer.startEditing()
```

Ezután írjuk meg a ciklust, amellyel egyenként végigmegyünk az elemeken. Mivel a rétegen lévő elemek *MultiLineString*ek, ezért két ciklus fog kelleni. Az egyikben elkapjuk az elemeket, a másikban a csoportgeometria (multigeometria) részein megyünk végig. Utána tudunk iterálni a részek (vonalszakaszok) csomópontjain. Az eredményt már egyszerűsítve *LineString*ként fogom kiírni. Amikor megvannak az egyes csomópontok, akkor a fenti egyenletekkel számítsuk ki az új vonalat. Az egyenletek célja, hogy miután kiszámítottuk a folytonos görbe kontrollpontjait, azok és a végpontok segítségével számítsuk ki magát a görbét.

```
for feat in features:
   for part in feat.geometry().asMultiPolyline():
        t=[]
        p='LINESTRING('
        if caps & QgsVectorDataProvider.AddFeatures:
            feature = QgsFeature()
            for pnt in part:
                t.append([pnt.x(), pnt.y()])

            p+=str(t[0][0])+' '+str(t[0][1])+', '

            for i in range(1, len(t)-2):

                xc1=(t[i-1][0]+t[i][0])/2.0
                yc1=(t[i-1][1]+t[i][1])/2.0
                xc2=(t[i][0]+t[i+1][0])/2.0
                yc2=(t[i][1]+t[i+1][1])/2.0
                xc3=(t[i+1][0]+t[i+2][0])/2.0
                yc3=(t[i+1][1]+t[i+2][1])/2.0

                len1=math.sqrt((t[i][0]-t[i-1][0])*(t[i][0]-t[i-
1][0])+(t[i][1]-t[i-1][1])*(t[i][1]-t[i-1][1]))
                len2=math.sqrt((t[i+1][0]-t[i][0])*(t[i+1][0]-
t[i][0])+(t[i+1][1]-t[i][1])*(t[i+1][1]-t[i][1]))
                len3=math.sqrt((t[i+2][0]-t[i+1][0])*(t[i+2][0]-
t[i+1][0])+(t[i+2][1]-t[i+1][1])*(t[i+2][1]-t[i+1][1]))
                k1=len1/(len1+len2)
                k2=len2/(len2+len3)
                xm1=xc1+(xc2-xc1)*k1
                ym1=yc1+(yc2-yc1)*k1
```

```
                xm2=xc2+(xc3-xc2)*k2
                ym2=yc2+(yc3-yc2)*k2
                ctrl1_x=xm1+(xc2-xm1)*smooth_value+t[i][0]-xm1
                ctrl1_y=ym1+(yc2-ym1)*smooth_value+t[i][1]-ym1
                ctrl2_x=xm2+(xc2-xm2)*smooth_value+t[i+1][0]-xm2
                ctrl2_y=ym2+(yc2-ym2)*smooth_value+t[i+1][1]-ym2

                p0x=t[i][0]
                p0y=t[i][1]
                p3x=t[i+1][0]
                p3y=t[i+1][1]


                for n in range (0,10):
                    k=n/10
                    px=(1-k)**3*p0x+3*k*((1-k)**2)*ctrl1_x+3*k*k*(1-
k)*ctrl2_x+k*k*k*p3x
                    py=(1-k)**3*p0y+3*k*((1-k)**2)*ctrl1_y+3*k*k*(1-
k)*ctrl2_y+k*k*k*p3y
                    p+=str(px)+' '+str(py)+', '

            p+=str(t[len(t)-1][0])+' '+str(t[len(t)-1][1])
            p+=')'

            feature.setGeometry(QgsGeometry.fromWkt(p))
            newlayer.dataProvider().addFeatures([feature])
```

A végén fogadjuk el a módosításokat, és az új ideiglenes réteget adjuk a projekthez. Frissítsük a nézetet.

```
newlayer.commitChanges()

QgsProject.instance().addMapLayer(newlayer)
newlayer.triggerRepaint()
```

A Bézier-görbék approximációs görbék, amely azt jelenti, hogy a kontrollpontokon nem halad át a görbe, csak megközelíti azokat. A Bézier-görbékből többféle létezik, a grafikus szoftverek leggyakrabban a másod- vagy harmadfokú görbékkel dolgoznak. Ebben a feladatban harmadfokú görbét fogunk számítani. A görbe görbeségét egy 0 és 1 közötti szám befolyásolja. A 0 a szögletes vonallal megegyezik, az 1 adja a legívesebb lefutást. Az feladatban 0.7-et választottam, de érdemes összehasonlítani pl. a 0, 0.5, 0.7 és 1-es értékek között a különbségeket.

45. feladat: 3D-s KML fájlok készítése népességi adatokból

45_ 3d_kml.py

A 13. feladatban kiszámoltuk a népsűrűséget az *mo.gpkg* megye rétegéhez. Az oszlopot nepsurnek neveztük. Most ebből a mezőből készítsünk 3D-s térképet!

A feladat magyarázatát úgy írtam meg, hogy feltételezem az olvasó többnyire tisztában a KML fájlok szerkezetével. Az ismeretek felelevenítéshez, pótlásához ajánlom egyszer a hivatalos dokumentációt:

https://developers.google.com/kml/documentation/kmlreference,

valamint Gede Mátyás tanár úr oktató anyagait:

https://mercator.elte.hu/~saman/hu/.

A feladatban KML-fájlokat fogok íratni, mint egy szöveges fájl. A QGIS bár tud KML fájlokat készíteni, de animációt és 3D-s adatokat még nehezebb hozzátenni a fájlokhoz, az alapszoftver erre egyelőre még nincs felkészülve.

Ebben a feladatban a népsűrűség értéke fogja adni a hasáb magasságát (illetve, ezt beszoroztam még százzal, egyébként nehezen mutatkoznak meg a különbségek a hasonló értékek között).

A KML fájlban háromféle stílust fogok definiálni. Ezek a felületek különböző színeiként nyilvánulnak meg a következő kategorizálással, pl. 0–90, 91–250, és 250 felett. A színek megadása hexadecimális értékekkel történik, pl. b03a2eff, ahol az utolsó két szám vagy betű az átlátszóságot adja meg.

Emellett a poligonokat 3D-ben kell megjelenítenem, ehhez elengedhetetlen, hogy a hosszúság, szélesség, mellett a magasságot is megadjuk. A magasság megadása *relativeToGround* rendszerben történik, vagyis a felszín felett értelmezi a harmadik koordinátát. Emellett kössük össze a felszínnel, ezt az *<extrude>1</extrude>* végzi el.

Ne feledkezzünk meg arról, hogy a KML csak földrajzi koordinátákkal dolgozik, ezért számítsuk át a koordinátákat a már tanult módon.

A feladatban lesz egy különleges helyzet, ez pedig Pest vármegye: a poligonnak gyűrűje van, a KML elkészítésénél eszerint kell eljárni.

Miután hozzáfértünk az aktív réteghez nyissunk meg egy szöveges fájlt írásra. Adjuk meg a KML fájllécét, majd írjuk meg a *Style* tageket (három poligon stílus van a fent említett három kategóriáknak).

```
layer=iface.activeLayer()
f=open(C:/adatok/könyvtára/3d.kml','w')
f.write('<?xml version="1.0" encoding="UTF-8"?><kml
xmlns="http://www.opengis.net/kml/2.2"
xmlns:gx="http://www.google.com/kml/ext/2.2"
xmlns:kml="http://www.opengis.net/kml/2.2" >')
f.write('<Document><name>Nepsuruseg.kml</name><open>1</open><Folder>')
f.write('<Style
id="kicsi"><LineStyle><color>f7dc6fff</color></LineStyle><PolyStyle><color>
f7dc6fff</color></PolyStyle></Style>')
f.write('<Style
id="kozepes"><LineStyle><color>f39c12ff</color></LineStyle><PolyStyle><colo
r>f39c12ff</color></PolyStyle></Style>')
```

```
f.write('<Style
id="nagy"><LineStyle><color>b03a2eff</color></LineStyle><PolyStyle><color>b
03a2eff</color></PolyStyle></Style>')
source_crs = QgsCoordinateReferenceSystem(layer.crs().authid())
target_crs = QgsCoordinateReferenceSystem("EPSG:4326")
```

Menjünk végig a réteg elemein. Nyissuk meg a *Placemark* elemet. A nevét olvassuk ki a név mezőből. A stílust határozzuk meg a magasságból. Majd kezdjük el az elemek geometriáját is megírni: poligon, az *altitudeMode relativeToGround, extrude* 1. Nyissuk meg a külső gyűrűt. Ezután transzformáljuk az elemet a 4326-os azonosítójú vetületbe.

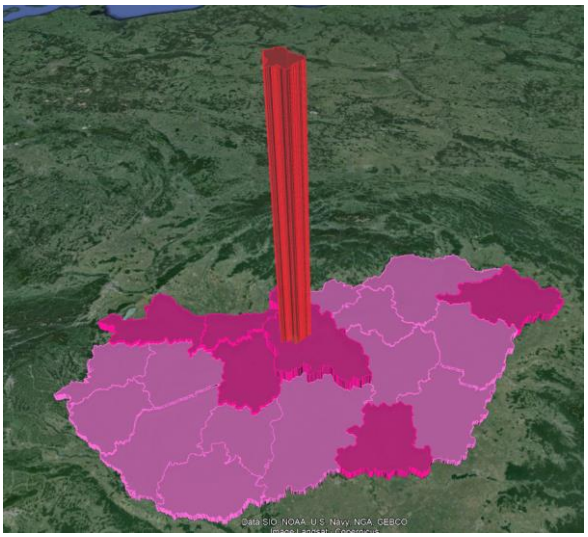
Ebben a fájlban ugyan nincsenek részek (poligoncsoportok, vagyis *parts()*), de a programot felkészítettem erre is. Ezután olvassuk ki a poligon gyűrűit a *boundary()* függvénnyel. Ebből Pest vármegyét kivéve mindenhol egy lesz, ezért ide egy elágazás kerül. Ahol csak egy külső gyűrű van, ott ennek töréspontjait kiegészítem a magassággal. Ezután lezárom a poligont. Pest megye esetében a külső gyűrű után szükség lesz egy *innerBoundary*-ra is. Csak ezután tudom lezárni a poligont.

```
for feature in layer.getFeatures():
    f.write('\n <Placemark>')
    f.write('<name>'+str(feature.attributes()[1])+'</name>')
    if feature.attributes()[4]<90:
        f.write('<styleUrl>#kicsi</styleUrl>')
    elif 90<feature.attributes()[4]<250:
        f.write('<styleUrl>#kozepes</styleUrl>')
    else:
         f.write('<styleUrl>#nagy</styleUrl>')
    mag=feature.attributes()[4]*100

f.write('<Polygon><tessellate>1</tessellate><altitudeMode>relativeToGround<
/altitudeMode><extrude>1</extrude><outerBoundaryIs><LinearRing><coordinates
>')
    geom = feature.geometry()
    geom.transform(QgsCoordinateTransform(source_crs, target_crs,
QgsProject.instance()))


    for part in geom.parts():
        bnum=part.childCount()
        b=part.boundary()
        if bnum==1:
            for p in part.vertices():
                f.write(str(p.x())+','+str(p.y())+','+str(mag)+' ')
            f.write('</coordinates></LinearRing></outerBoundaryIs>')
        else:
            gyuru=1
            if gyuru==1:
                for p in b[0].vertices():
                    f.write(str(p.x())+','+str(p.y())+','+str(mag)+' ')
                f.write('</coordinates></LinearRing></outerBoundaryIs>')
                gyuru=gyuru+1
            else:
                f.write('<innerBoundaryIs><LinearRing><coordinates>')
                for p in b[1].vertices():
                    f.write(str(p.x())+','+str(p.y())+','+str(mag)+' ')
```

```
                f.write('</coordinates></LinearRing></innerBoundaryIs>')
        f.write('</Polygon></Placemark> \n')
```

Végül lezárom a KML-t a megfelelő tagekkel, és bezárom a szöveges fájlt is.

```
f.write('</Folder></Document></kml>')
f.close()
```

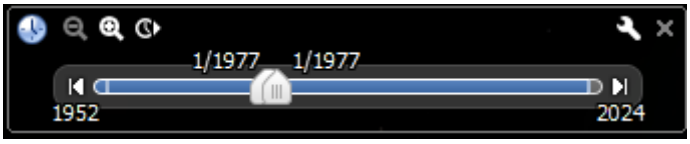


46. feladat: KML fájlok: animáció készítése

45_ kml_animation.py

Készítsünk egy animációt KML-ben, amely az Európai Unió keletkezését mutatja be. Használjuk hozzá az *eu.shp* fájl, ebben benne vannak a csatlakozási és kilépési időpontok.

Az animáció helyes lejátszásához a Google Earth animációs munkaablakában kössük össze a gomb és félhold alakú jelölőket!



Az előző feladathoz hasonlóan ennek magyarázata is úgy készült el, hogy feltételezi az olvasó alapvető KML ismereteit. Ajánlott szakirodalom az előző fejezetben van.

Férjünk hozzá az aktív réteghez. Nyissunk meg egy szöveges fájlt, adjuk meg a KML fejlécét. Az országok kitöltése most azonos lesz. Ne felejtsük el megadni a kiindulási (réteg vetülete), és a célvetületet, EPSG: 4326.

```
layer=iface.activeLayer()
f=open(C:/adatok/könyvtára/anim.kml','w')
f.write('<?xml version="1.0" encoding="UTF-8"?><kml
xmlns="http://www.opengis.net/kml/2.2"
xmlns:gx="http://www.google.com/kml/ext/2.2"
xmlns:kml="http://www.opengis.net/kml/2.2" >')
f.write('<Document><name>Nepsuruseg.kml</name><open>1</open><Folder>')
f.write('<Style
id="orszag"><LineStyle><color>b03a2eff</color></LineStyle><PolyStyle><color
>b03a2eff</color></PolyStyle></Style>')
source_crs = QgsCoordinateReferenceSystem(layer.crs().authid())
```

```
target_crs = QgsCoordinateReferenceSystem("EPSG:4326")
```

Menjünk végig a réteg elemein. Hozzuk létre a *Placemark*ot. Adjuk meg a *name* tulajdonságot az országnév mezőből. Adjuk meg a *TimeSpan* animáció kezdeti és záró dátumát (<*begin*> és <*end*> *tag*ek). A záró dátumnak, ahol 0 szerepel vegyük az aktuális évet, pl. 2024-et. A 0 azt jelenti, hogy az ország jelenleg is tagja az Uniónak.

Adjuk hozzá a *styleUrl*-nél az országok színezését.

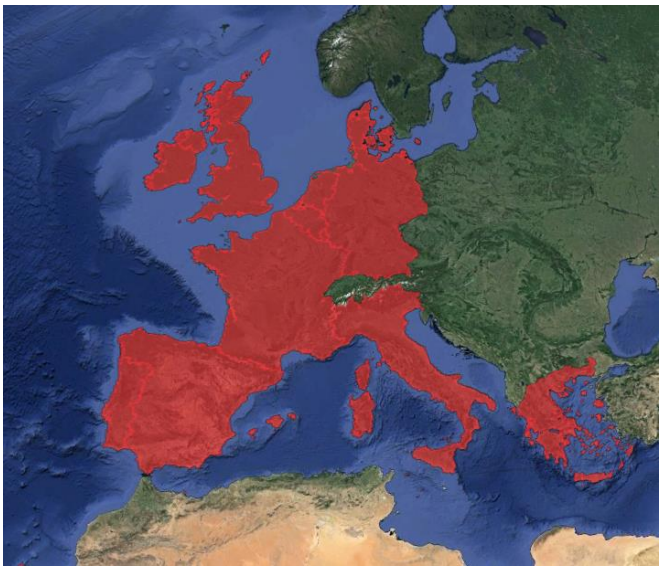
```
for feature in layer.getFeatures():
    f.write('\n <Placemark>')
    f.write('<name>'+str(feature.attributes()[0])+'</name>')
    if feature.attributes()[2]==0:
        fa=2024
    else:
        fa=feature.attributes()[2]

f.write('<TimeSpan><begin>'+str(feature.attributes()[1])+'</begin><end>'+st
r(fa)+'</end></TimeSpan>')

    f.write('<styleUrl>#orszag</styleUrl>')
```

Mivel az országok többsége *MultiPolygon*ként van megadva (csoportgeometria), ezért itt is ezt kell alkalmazni. Jelenleg a *MultiGeometry* helyettesíti bármely csoportgeometriát, azon belül lehetnek pontok, vonalak vagy poligonok.

Menjünk végig a poligoncsoportokon, vagyis a részeken. Minden résznél új poligont nyitunk meg. A geometriákat alakítsuk át 4326-os azonosítójú vetületbe, használjunk földrajzi koordinátákat. Ha a poligonnak vannak gyűrűi, azokat is tegyük bele (mint az előző feladatban <*innerBoundary*>-ként).

```
    f.write('<MultiGeometry>')
    geom = feature.geometry()
    geom.transform(QgsCoordinateTransform(source_crs, target_crs,
QgsProject.instance()))

    resz=0
    for part in geom.parts():
        f.write('<Polygon><tessellate>1</tessellate><altitudeMode>
clampToGround</altitudeMode><outerBoundaryIs><LinearRing><coordinates>')
        bnum=part.childCount()
        b=part.boundary()
        if bnum==1:
            for p in part.vertices():
                f.write(str(p.x())+','+str(p.y())+',0 ')
            f.write('</coordinates></LinearRing></outerBoundaryIs>')
        else:
            gyuru=1
            if gyuru==1:
                for p in b[0].vertices():
                    f.write(str(p.x())+','+str(p.y())+',0 ')
                f.write('</coordinates></LinearRing></outerBoundaryIs>')
                gyuru=gyuru+1
            else:
                f.write('<innerBoundaryIs><LinearRing><coordinates>')
                for p in b[1].vertices():
```

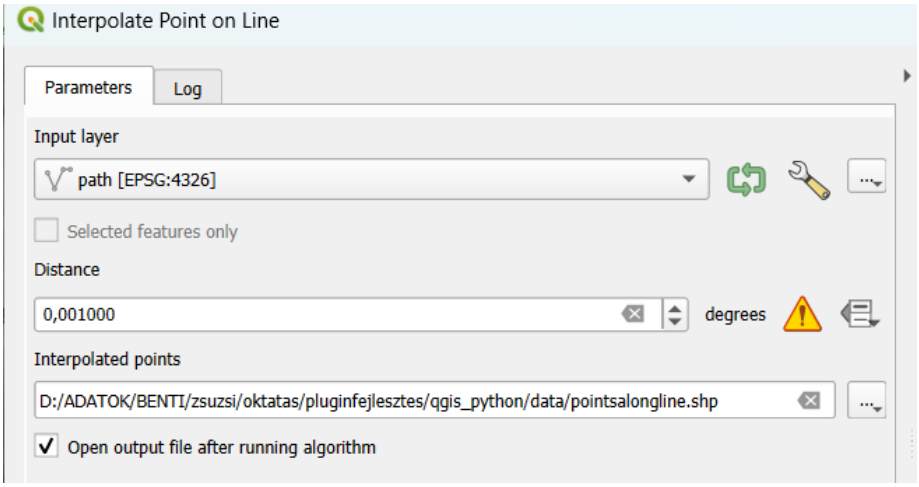
```
                    f.write(str(p.x())+','+str(p.y())+',0 ')
                f.write('</coordinates></LinearRing></innerBoundaryIs>')
        f.write('</Polygon> \n')
    f.write('</MultiGeometry></Placemark>')
f.write('</Folder></Document></kml>')
f.close()
```

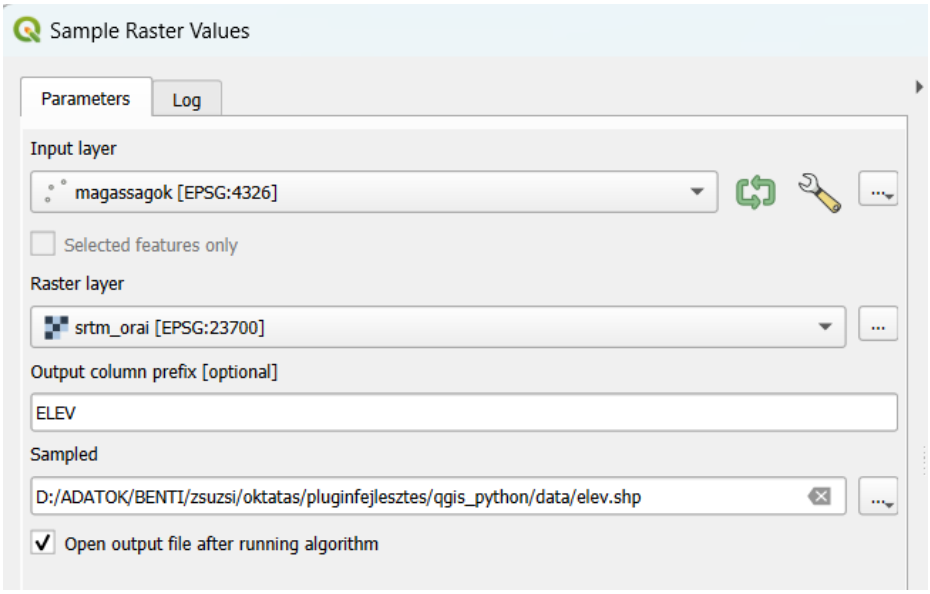


47. feladat: Készítsünk egy szkriptet, amely kiolvassa egy túraútvonal mentén a magasságokat

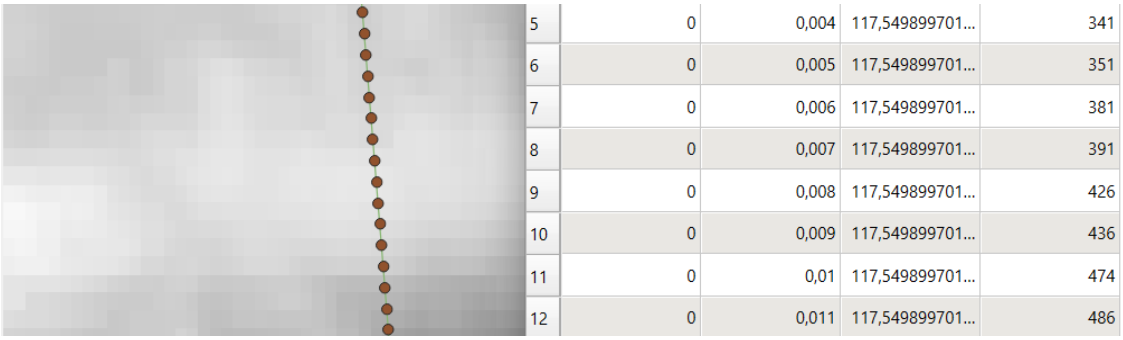
47_ raster_vector.py

A szkript célja, hogy a *Processing* eszközöket felhasználva és egymás után fűzve létrehozza az eredményréteget. A feldolgozáshoz raszteres és vektoros rétegeket is felhasználunk. A feladathoz szükség lesz egy domborzatmodellre, például azt *srtm_orai.tif*-re, és rajzoljunk vagy töltsünk le valahonnan egy létező túraútvonalat, amely a Mátra ezen részén vezet keresztül. Szükségem lesz a túraútvonal magassági profiljára, vagyis írassuk bizonyos távolságonként a domborzatmodellből a magasságokat. Ehhez segítségül két eszközt tudok használni, az egyik az *Interpolate point on line*, a másik pedig a *Sample raster values*, amellyel a képzett pontokhoz a raszterből hozzárendelhető a magasság. A képek a beállításokat mutatják.





Az alábbi képen pedig az eredmény látszik. A táblázat utolsó oszlopa tartalmazza a magasságokat, amelyeket a raszteres képből vettünk át.



5	0	0,004	117,549899701...	341
6	0	0,005	117,549899701...	351
7	0	0,006	117,549899701...	381
8	0	0,007	117,549899701...	391
9	0	0,008	117,549899701...	426
10	0	0,009	117,549899701...	436
11	0	0,01	117,549899701...	474
12	0	0,011	117,549899701...	486

A két algoritmus egymás utáni lefuttatásával nyerhető ki a végeredmény, majd adjuk hozzá az eredményt a projekthez.

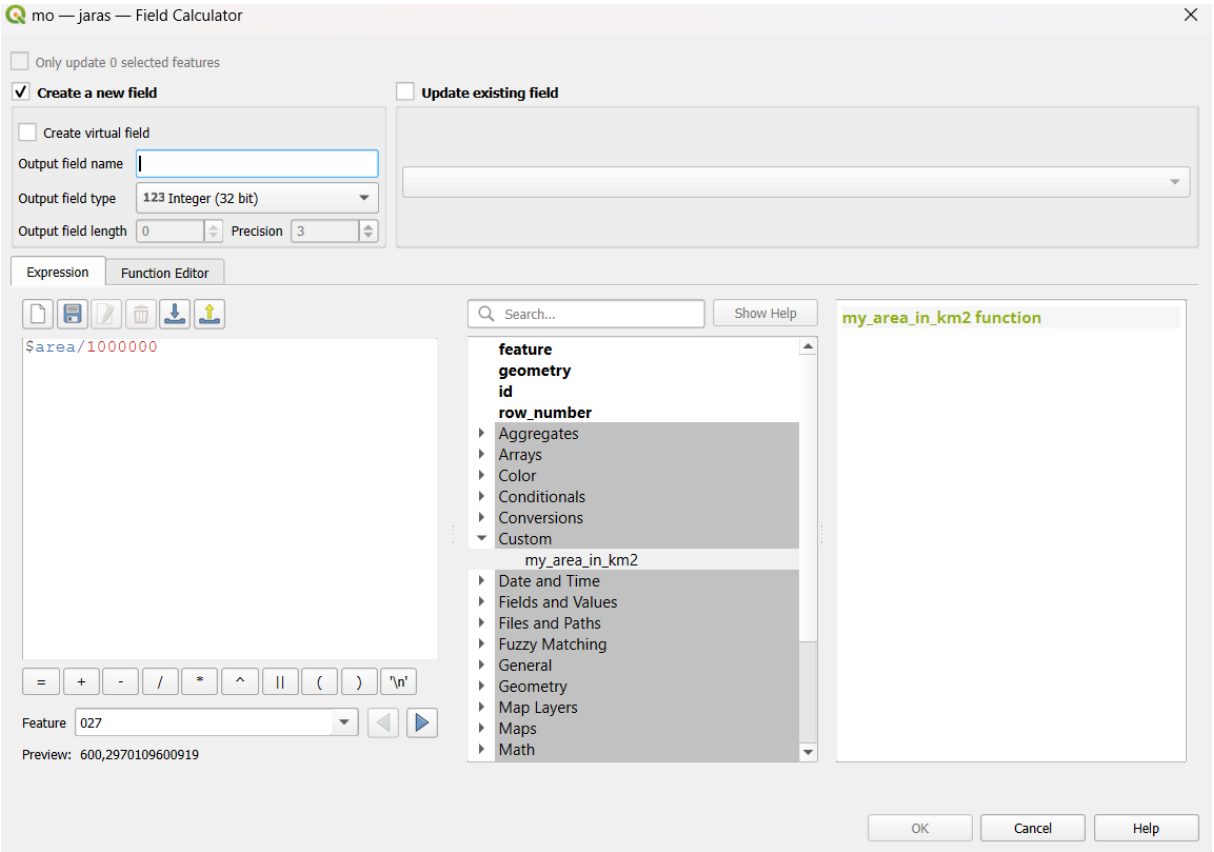
```
processing.run("native:pointsalonglines",
{'INPUT':'C:/adatok/forrása/path.shp',
'DISTANCE':0.001,
'START_OFFSET':0,
'END_OFFSET':0,'
OUTPUT': C:/adatok/ forrása/pointsalongline.shp'})

processing.run("native:rastersampling", {
'INPUT': C:/adatok/forrása/pointsalongline.shp',
'RASTERCOPY':' C:/adatok/ forrása /srtm_orai.tif',
'COLUMN_PREFIX':'ELEV',
'OUTPUT': C:/adatok/ forrása/elev.shp'})

path= C:/adatok/ forrása/elev.shp'
layer=QgsVectorLayer(path,"magassagok","ogr")
if not layer.isValid():
    print("Layer failed to load!")
else:
    QgsProject.instance().addMapLayer(layer)
```

48. feladat: Saját SQL függvény definiálása

48_ sql_query.py

Ha többször kell ismétlődő műveleteket kell végrehajtanunk állományainkon, akkor érdemes lehet előre megírt függvényekben gondolkozni. Például, gyakran lehet arra szükség, hogy egy elem területét kiszámítsuk, de nem négyzetméterben, hanem négyzetkilométerben. Ekkor, hogy ne kelljen annyit írni, célszerű készíteni és elmenteni az erre szolgáló kifejezést. Ez igaz lehet minden olyan esetre, amelyben összetett függvényeket írunk, és ezeket nem akarjuk újra és újra begépelni.

Nézzük az előbb felhozott példát. Hozzunk létre egy új oszlopot, azt töltsük fel a terület méretével km^2-ben.



Ha el szeretnénk menteni a függvényt, menjünk a *Function Editor* fülre. Hozzunk létre a zöld keresztre kattintva egy új függvényt, pl. *terulet* néven. A függvény kifejezése a következő lesz:

```
from qgis.core import *
from qgis.gui import *

@qgsfunction(group='Custom', referenced_columns=[])
def my_area_in_km2(area):
    return area/1000000
```

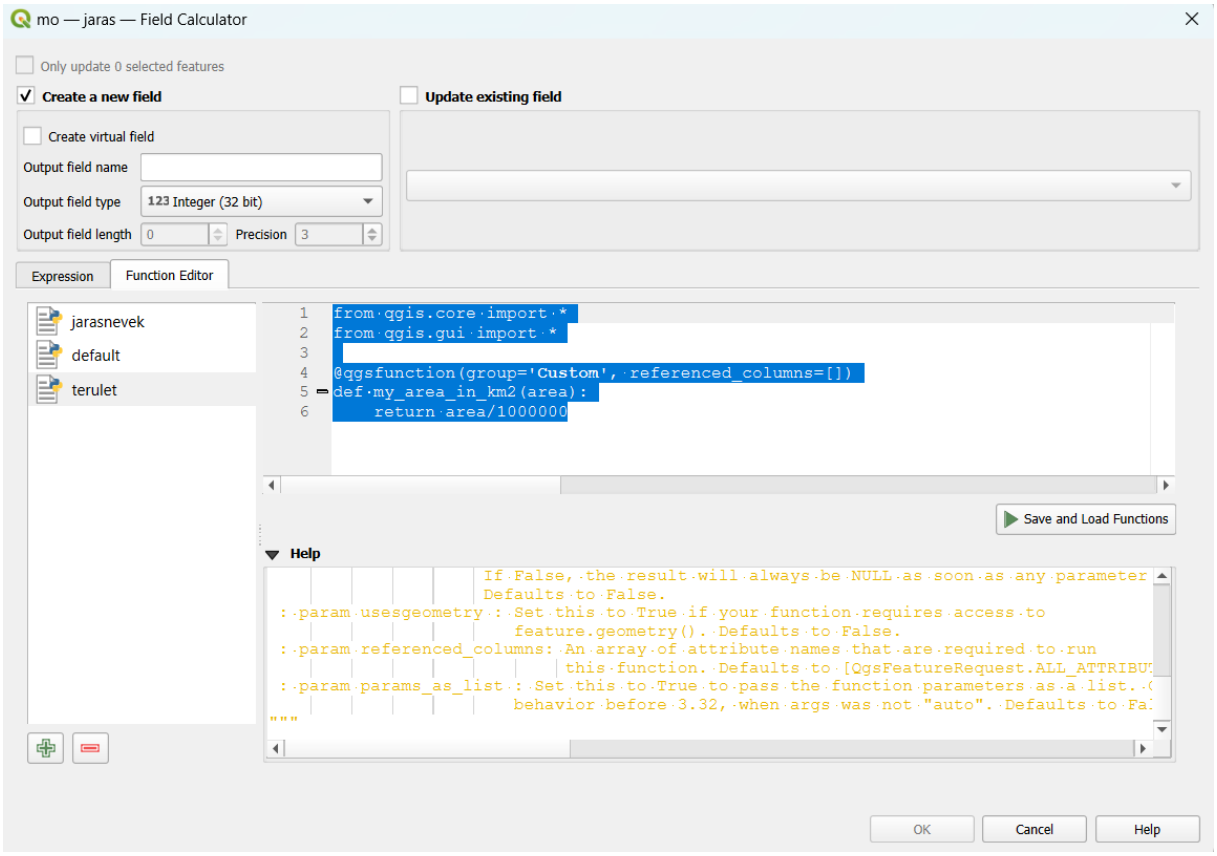
A *@qgsfunction* részben lehet megmondani, hogy melyik legördülő listába kerüljön a függvény, ez most a *Custom*. A függvény neve *my_area_in_km2*, egy bemeneti paramétert vár, ez az elem területe, vissza pedig az elem km^2-ben vett területét adja meg.

A függvény meghívása pedig az *Expression* fülön a következőképpen történik.

```
my_area_in_km2($area)
```

Mellékelek egy képet a *Function Editor* fülről is. A mentett függvény egyébként a

C:\Users\USERNAME\AppData\Roaming\QGIS\QGIS3\profiles\default\python\expressions mappába kerül, mint egy Python szkript.



Készítsünk egy függvényt, hogy a *mo.gpkg* állomány település rétegén megnézzük, melyik település melyik százezres szelvényre esik. Írjuk be a százezres szelvény számát az attribútum táblázatba.

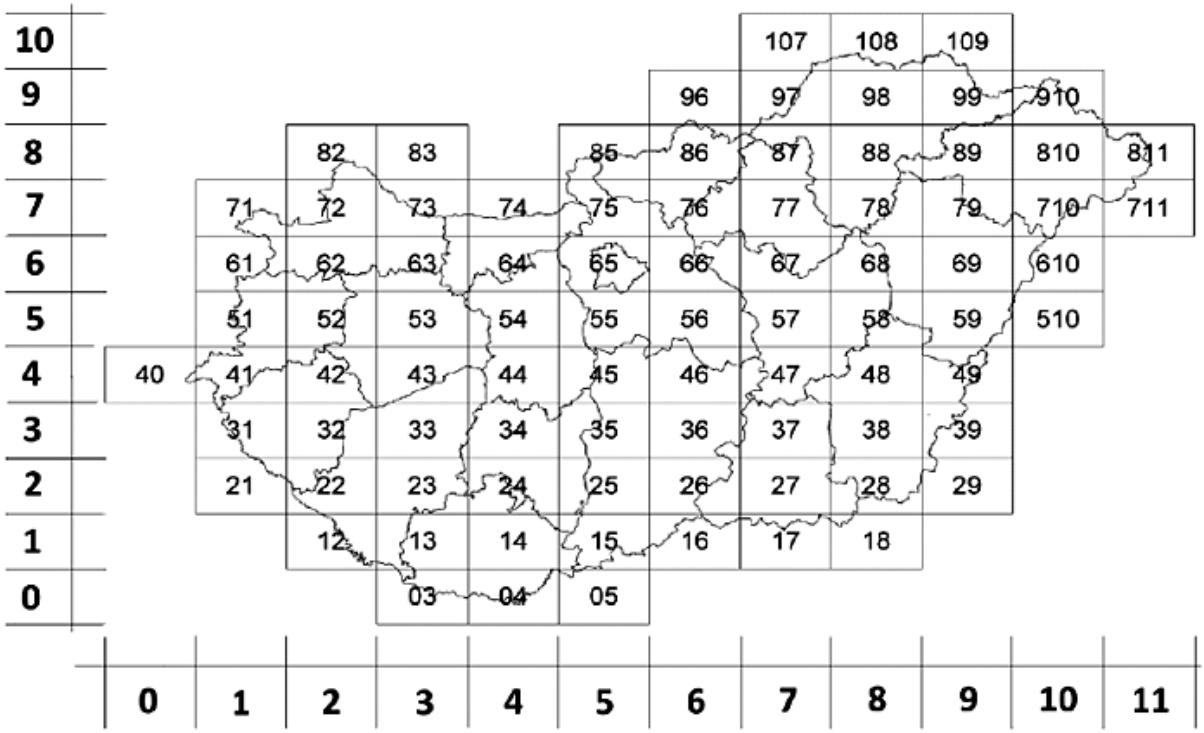
A függvény neve legyen *eotr_szelvenykereso_100e(x,y)* és a két paraméter a település x és y koordinátája, ebben a sorrendben megadandó. A mező, amelyet létrehozunk legyen szöveges adattípus (egész szám azért nem jó, mert ha a szelvényszám 0-val kezdődik nem tudjuk megjeleníteni). Ha meg akarjuk hívni a függvényt, csak a két koordinátát kell kitölteni: *eotr_szelvenykereso_100e($x,$y).*

Az első dolgunk az, hogy definiáljuk a százezres szelvények határait. Ha nincs kézreeső áttekintő szelvény pontos koordinátákkal, vegyünk elő egy százezres szelvényt, és számítsuk ki abból a határoló koordinátákat. Például vegyük a 65-ös, Budapestet és környékét ábrázoló szelvényt. A határoló koordináták (x) 624000 m és 672000 m és (y) 224000 m és 256000 m. Ebből következik, hogy a szelvény 48000 méter széles (x) és 32000 m magas (y). A szelvényszám a 65, a 6. sort, és 5. oszlopot jelenti. Így ki tudjuk számítani az egész országra a határoló koordinátákat, lásd lentebb.

Ezek után két for ciklus szükséges, amelyben megvizsgálom, hogy az aktuális település x vagy y koordinátája melyik két százezres tengely közé esik.

$$y_{min} + i * y_{step} \leq sor \leq y_{min} + (i+1) * y_{step}$$

$$x + j * x_{step} \leq oszlop \leq x_{min} + (j+1) * x_{step}$$



Végül összefűzzük a két számot, mint szöveg.

```
from qgis.core import *
from qgis.gui import *

@qgsfunction(group='Custom', referenced_columns=[])
def eotr_szelvenykereso_100e(x,y):
    minx=384000
    maxx=960000
    stepx=48000
    miny=32000
    maxy=384000
    stepy=32000
    row=0
    col=0
    for i in range(0,13):
        if (miny+i*stepy)<=y and (miny+(i+1)*stepy)>=y:
            row=i

    for j in range(0,12):
        if (minx+j*stepx)<=x and (minx+(j+1)*stepx)>=x:
            col=j

    sheetnumber=str(row)+str(col)
    return sheetnumber
```

	fid	nev	ksh_kod	jogallas	terulet_ha_2018	nepesseg_2018	Y_eov	sz3
1	648	Aba	17376	város	8805	4416	187361	44
2	974	Abádszalók	12441	város	13223	4060	238057	67
3	33	Abaliget	12548	község	1609	644	88917	14
4	335	Abasár	24554	község	2082	2564	272763	77
5	2097	Abaújalpár	15662	község	848	49	331510	98

A QGIS visszacsatoló üzenetei

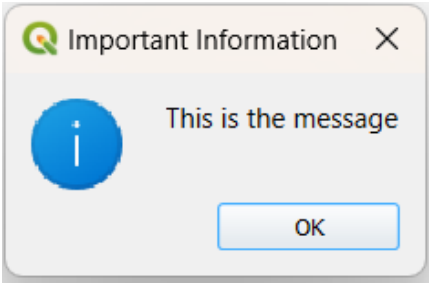
49. feladat: A QGIS visszacsatoló üzenetei és létrehozásuk

49_ feedbacks_and_messages.py

A QGIS-ben lehetőség van a felhasználóval kommunikálni üzenetek formájában. Ez lehet előreugró ablak (*MessageBox*), a térképi vászon felső sávjában megjelenő üzenetküldő sáv (*MessageBar*). Ebben a feladatban erre hoztam néhány jellemző, gyakran használt példát.

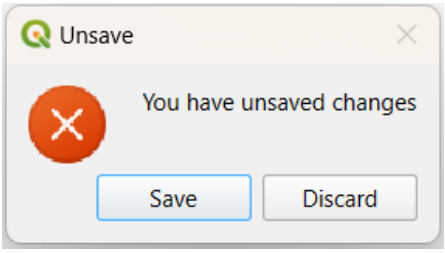
Az előreugró ablak a *PyQt5* modul része, azt külön importálni szükséges. Ezután, ha figyelmeztető üzetenet szeretnénk küldeni *QMessageBox.information* függvénnyel tehető meg. A képen látható, hogy a kódba beírt paraméterek mely részén jelennek meg az ablaknak.

```
from PyQt5.QtWidgets import QMessageBox
QMessageBox.information(iface.mainWindow(),'Important Information','This is the message')
```



A *MessageBox* lehet figyelemfelkeltő is, ez a *critical* függvénnyel érhető el. Itt a lehetséges válaszok száma már kettő, például egy mentés vagy egy elvetés.

```
QMessageBox.critical(None, "Unsave", "You have unsaved changes", QMessageBox.Discard|QMessageBox.Save)
```



Ezeken kívül *warning* (figyelmeztetés) és *question* (kérdés) ablakokat tudunk létrehozni.

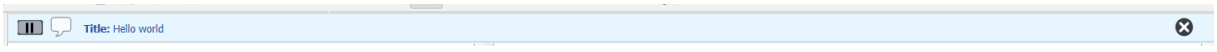
Ennél gyakrabban alkalmazzuk, hogy a térképi vászon felső sávjában osztunk meg üzeneteket a felhasználóval – ez a *messageBar()*. Ez a QGIS interfész része, nem kell hozzá importálni semmit. Ennek is négy fajtája van, mindre hoztam egy-egy példát.

A figyelmeztetés (*Warning*) sárga háttérrel jelenik meg, és akkor használatos, ha valamit nem tud úgy végrehajtani a modul, ahogy szeretné a felhasználó.

```
iface.messageBar().pushMessage('Title', 'Hello world', Qgis.Warning,5)
```

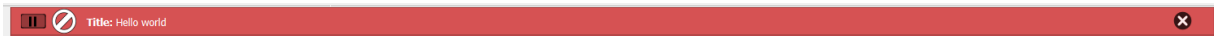
Az információ (*Info*) például egy folyamat elindulásáról tudósítja a felhasználót.

```
iface.messageBar().pushMessage('Title', 'Hello world', Qgis.Info,5)
```



A hibaüzeneteket (*Critical*) vörös háttérrel jelenítjük meg, és a folyamat sikertelenségéről tudósít.

```
iface.messageBar().pushMessage('Title', 'Hello world', Qgis.Critical,5)
```



A zöld hátterű visszajelzés (*Success*) például egy sikeres szkriptlefutásról tudósítja a felhasználót.

```
iface.messageBar().pushMessage('Title', 'Hello world', Qgis.Success,5)
```

Mindegyik paraméterezése a következőképpen néz ki. Az első paraméter a cím, a második a hosszabb üzenet, és a harmadik a *messageBar* típusa. Az utolsó paraméter mondja meg, hány másodpercig látszódjon az üzenet, ennek érdemes legalább 5-10 másodpercet hagyni, főleg, ha hosszabb üzenetet szeretnénk közölni a felhasználóval. Ezután automatikusan bezárja a program az üzenetet.

Ezen kívül, lassabb, több időt igénybe vevő folyamatok esetén érdemes egy folyamat előrehaladását jelző információs sávot (*progressBar*) is feltenni a felhasználónak, amellyel nyomon követheti a folyamat állapotát.

Ezt kétféleképpen is megvalósíthatjuk. Vegyünk egy nagyobb adattartalmú réteget, pl. a települések az *mo.gpkg*-ból. Írassunk ki valamit a Python konzol segítségével, pl. egy pontot írjon ki minden elemnél (ez kellően időigényes folyamat).

Két helyen tudunk tudósítani a folyamatról, az egyik a lábléc (*statusBar* – ahol a koordináták, méretarány stb. találhatók) vagy a térképi vászon felső része egy *progressBar* formájában.

Nézzük az elsőt, a láblécet. Menjünk végig a réteg elemein. Nézzük meg, hány elem van a rétegen, és az hány százaléka a réteg összes elemének. Írjuk ezt ki a láblécben *statusBarIface().showMessage()* függvénnyel. A végén szüntessük meg az üzenetet (*statusBarIface().clearMessage()*).

```
layer = iface.activeLayer()
count = layer.featureCount()
features = layer.getFeatures()
for i, feature in enumerate(features):
    print('.') # printing should give enough time to present the progress
    percent = i / float(count) * 100
    iface.statusBarIface().showMessage("Processed {}
%".format(int(percent)))
iface.statusBarIface().clearMessage()
```

Ennél jobban észrevehető, ha ugyanezt egy *progressBar*on jelenítjük meg. Férjünk hozzá a réteg elemeihez, hívjuk meg a *messageBar()*-t. Definiáljuk a *QProgressBar()*-t, adjuk meg a helyzetét. Adjuk hozzá a *progressBar*-t, a *messageBar*-hoz. Majd menjünk végig a réteg elemein, számítsuk ki, hányadik elemnél járunk (%-ban, hasonlóan az előzőhöz), ezt a százalékot állísuk be a *setValue()* függvénnyel. Ha végeztünk az elemekkel, töröljük az eszközt.

```
layer = iface.activeLayer()
count = layer.featureCount()
features = layer.getFeatures()
progressMessageBar = iface.messageBar().createMessage("Doing something")
progress = QProgressBar()
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
```

```
iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)

for i, feature in enumerate(features):
      percent = i / float(count) * 100
progress.setValue(percent)
iface.messageBar().clearWidgets()
```

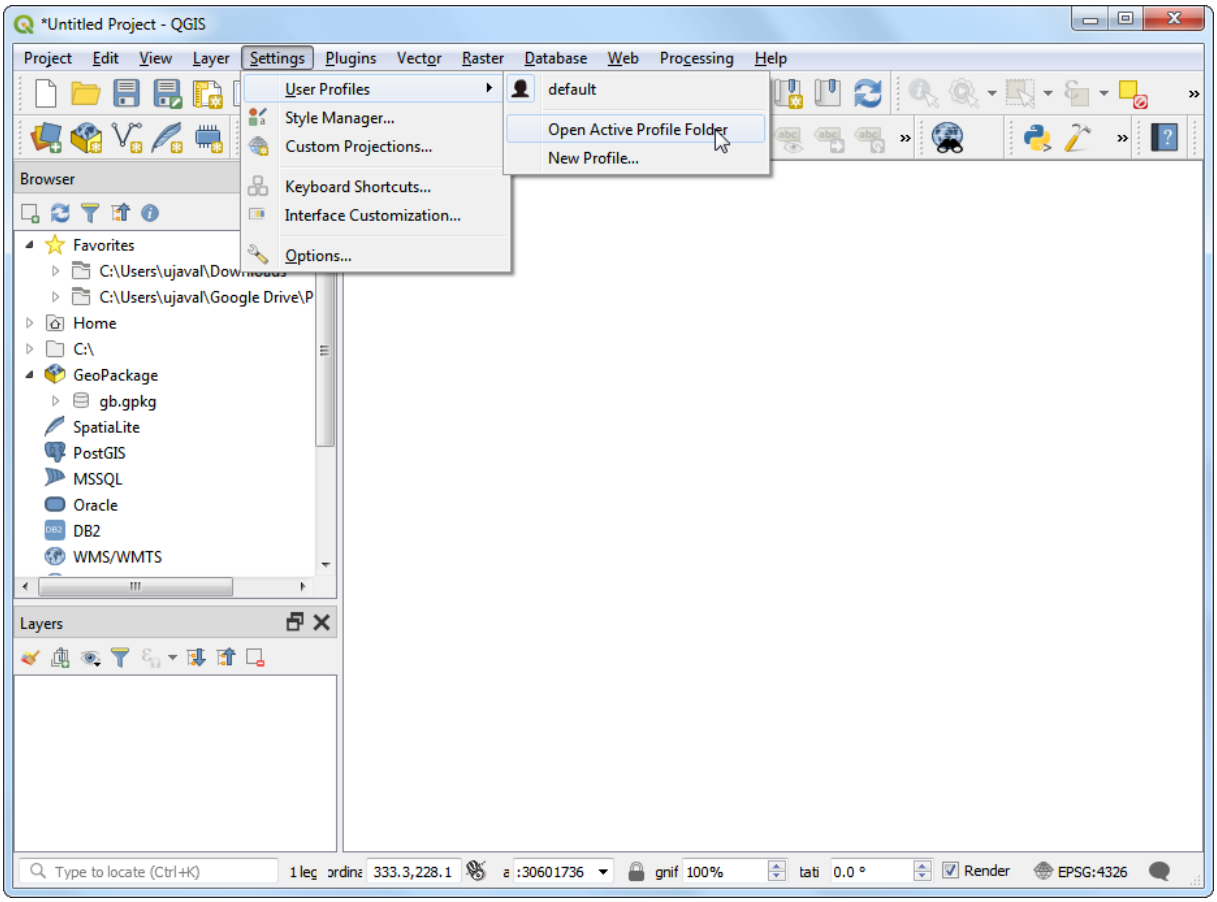
A pluginfejlesztés kezdő lépései

A pluginek építésének kétféleképpen lehet nekiállni. Minden pluginhez vannak kötelező fájlok, azokat a fejlesztő hozza létre. Azonban, ha kevésbé vagyunk gyakorlottak, talán kényelmesebb egy pluginfejlesztést segítő plugint alkalmazni, ez pedig a Plugin Builder. Telepítsük a tárházból.

A pluginfejlesztéshez ezt a weboldalt tudjuk segítségül hívni. https://plugins.qgis.org/

A pluginek a QGIS speciális mappájába kerülnek. Ez a mappa megnyitható a *Settings* → *User Profiles* → *Open Active Profile Folder* menüvel, vagy keressük meg a HOME könyvtárunkban az *AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins* mappát.

Megj: Ha nem a default profilt használjuk, akkor a kívánt profilnévben érdemes keresni.

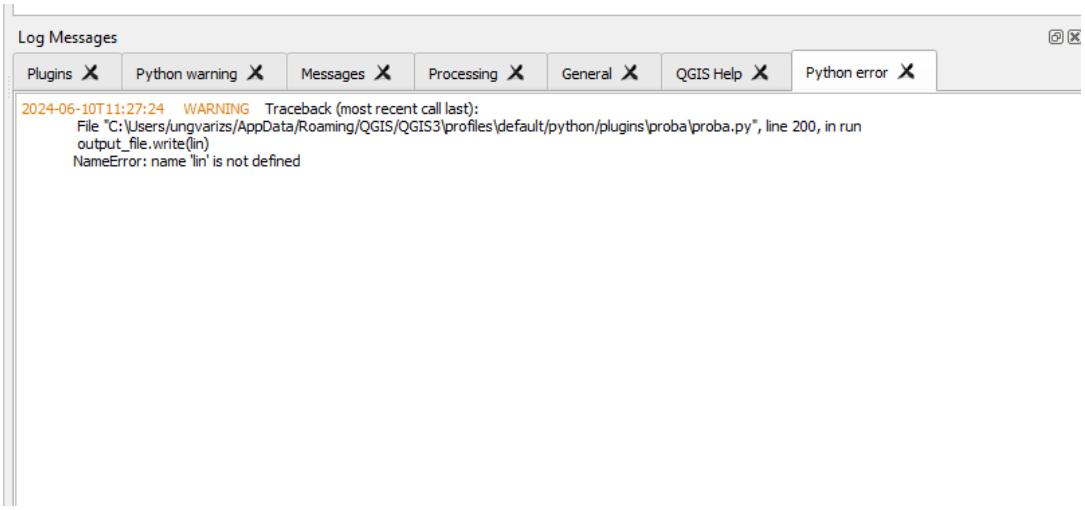


Néhány gondolat a pluginfejlesztésről

A pluginfejlesztés során továbbra is Python szkripteket fogunk írni, ezért szinte bármilyen kódszerkesztőt tudunk használni. Javaslom, hogy abban dolgozzon, amelyet már jól ismer. Talán a legkézenfekvőbb a QGIS beépített Python konzolját használni, így nem kell annyit váltani az ablakok között. Ha valaki ezzel nincs kibékülve, használhatja a Python IDLE környezetét (QGIS-szel szinte megegyező), vagy a PyCharm professzionális környzetetét. Esetleg egy Notepad++ kódszerkesztő is megfelelő stb.

A hibakeresés és a futtatás kissé módosul. Érdemes a tárházból letölteni a Plugin Reloader plugint, amely arra jó, hogy újratölti a kijelölt plugint. Tehát ha módosításokat végzünk a pluginhez tartozó bármely szkripten, nem kell becsukni és újra megnyitni a QGIS-t, hanem elegendő az éppen szerkesztett plugint újratölteni. Csak azokat a plugineket tudja újratölteni,

amelyeket az indításkor már inicializált: vagyis, ha az adott QGIS munkamenet közben hozunk létre egy új plugint, akkor azt nem látja. Ilyenkor a QGIS újraindítása szükséges, utána már tudjuk alkalmazni az újratöltést. A hibakereséshez használjuk a *View → Panels → Log Messages Panel* ablakot. Ebben a panelben több fül is van, ezenkben nyomon követhetjük a QGIS működését: *Plugins, Python warnings, Messages, Processing, General, QGIS Help, Python error*. Például egy változó elgépelése esetén a *Python error* fülön találunk információt a hiba pontos okáról és helyéről.



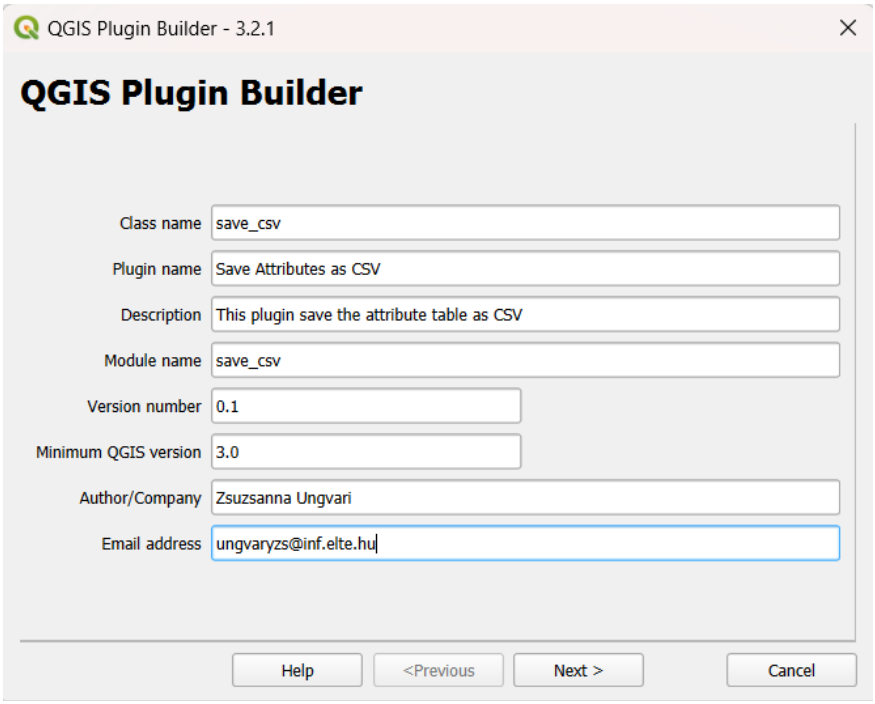
50. feladat: Készítsünk egy plugint, amely az attribútum táblát CSV-ként írja ki

Készítsünk saját modult a QGIS alá. Használjuk ehhez a *Plugin Builder* modult. Az új modul segítségével az egyes rétegek attribútum tábláit CSV fájlként tudjuk menteni. (Erre egyébként van lehetőség a QGIS-ben is.)

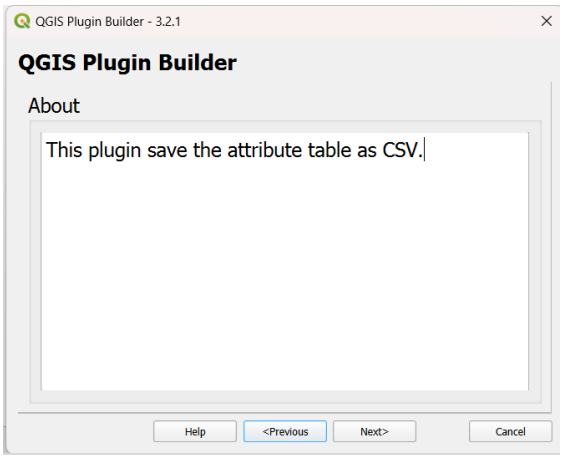
Ez a fejezet Ujaval Gandhi leckéje alapján készült.

http://www.qgistutorials.com/en/docs/building_a_python_plugin.html

A *Plugin Builder* elindítása után töltsük ki az alapadatokat. Adjuk meg az osztály, a plugin nevét és a modul nevét. Lehetőleg az osztály és a modul neve ne tartalmazzon ékezetes betűket. A rövid leírásban adhatjuk meg a plugin célját (ez a tárházban félkövér betűvel kiemelt rövid szöveg, lehetőleg angol nyelven). Emellett megadhatjuk a plugin verziószámát, a minimum elvárt QGIS verziót, a szerző nevét és email címét.

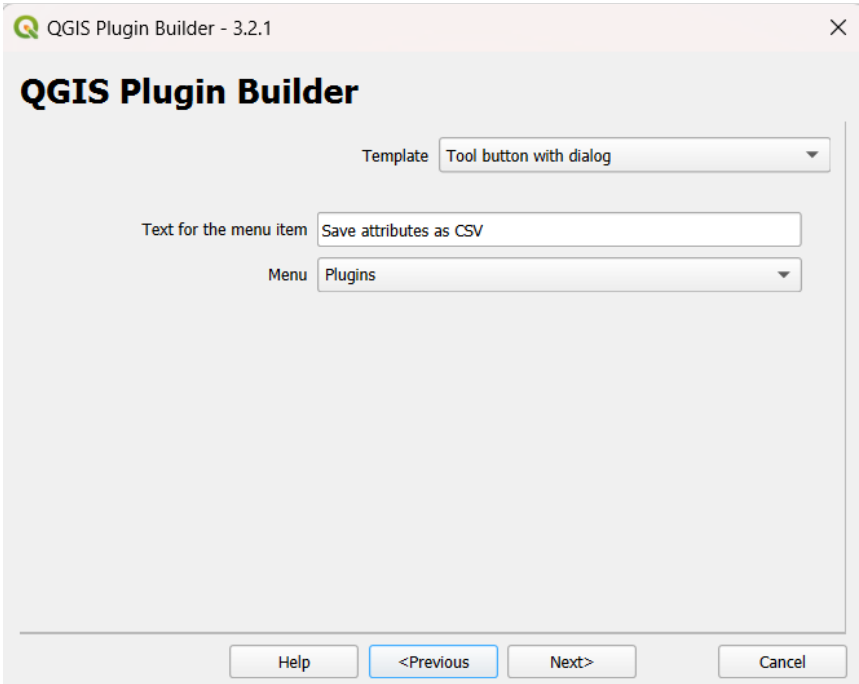


Az *About*-ban hosszabb leírás adható meg a pluginról, pl. mi változott a pluginben, milyen hibajavítás történt stb.



Ezek után tudjuk beállítani a plugin kezelőfelületének típusát. Válasszuk mintának (*template*) a *Tool button with dialog* módot.

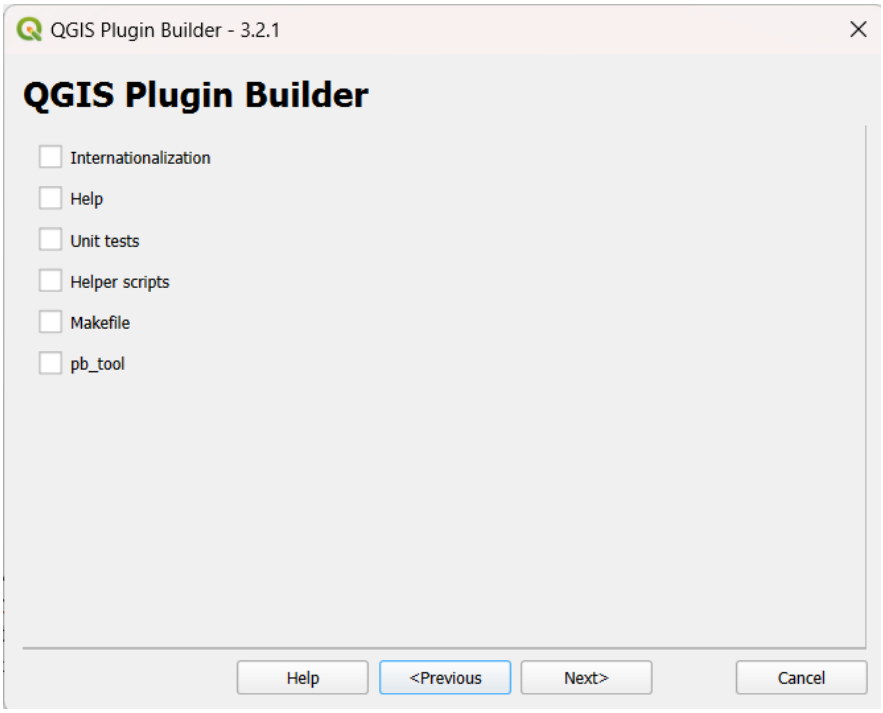
A Menu-nél kiválasztható, melyik Főmenübe kerüljön bele, legyen ez most a Plugins.



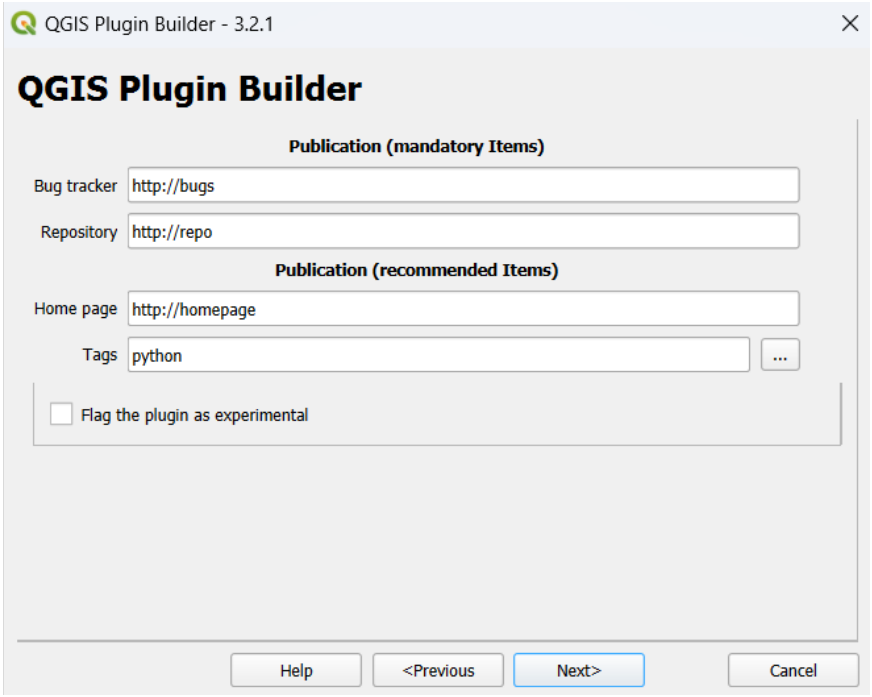
A következő lépésben megadható, milyen fájlokat készítsen el. A segédfájlok a következők lehetnek: *Internationalization* –többnyelvűsítés, *Help* – HTML segédlet, *Unit tests* – teszteléshez szükséges eszközök, *Helper scripts* – főleg a tárházban való publikálást segíti, *Makefile* – egy *Makefile*-t készít, amellyel GNU-val[1] vezérelhető lesz a fájl és a *pb_tool* –egy Python parancssoros eszköz a QGIS pluginok fordításához és telepítéséhez Linuxon, Mac OS X-en és Windowson).

Én most ezek nélkül fogom létrehozni a plugint, hogy minél kevesebb nem használt fájl legyen.

[1] GNU: A GNU Make egy olyan eszköz, amely a program forrásfájljaiból vezérli a program végrehajtható és egyéb, nem forrásfájljainak létrehozását. A Make a makefile nevű fájlból szerzi a program összeállításának módjára vonatkozó ismereteit, amely felsorolja az egyes nem forrásfájlokat, és azt, hogy azokat hogyan kell más fájlokból létrehozni. Amikor programot írunk, ha készül hozzá egy makefile, a Make segítségével a programot össze lehet állítani és telepíteni. https://g-sherman.github.io/Qgis-Plugin-Builder/#using-the-makefile

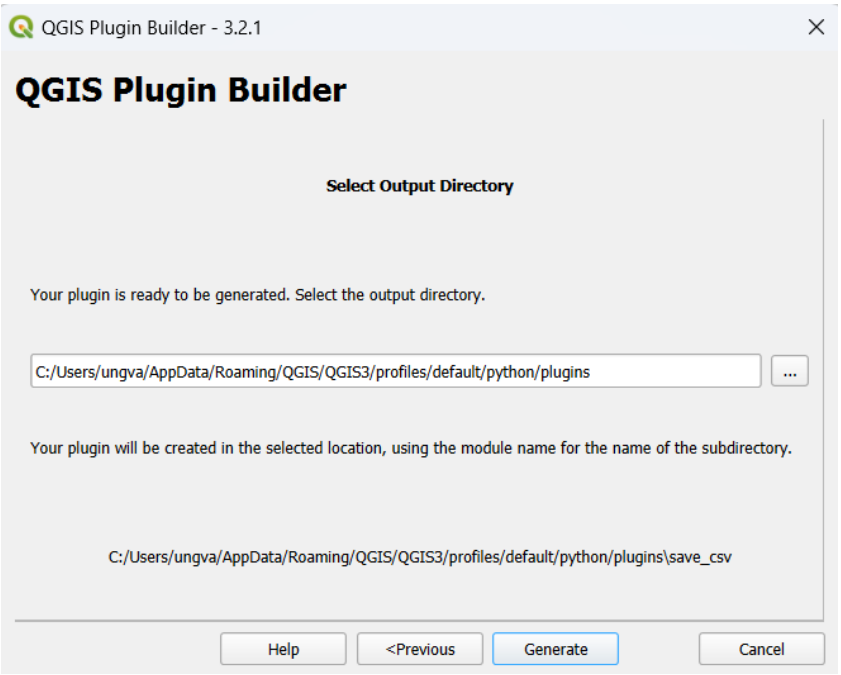


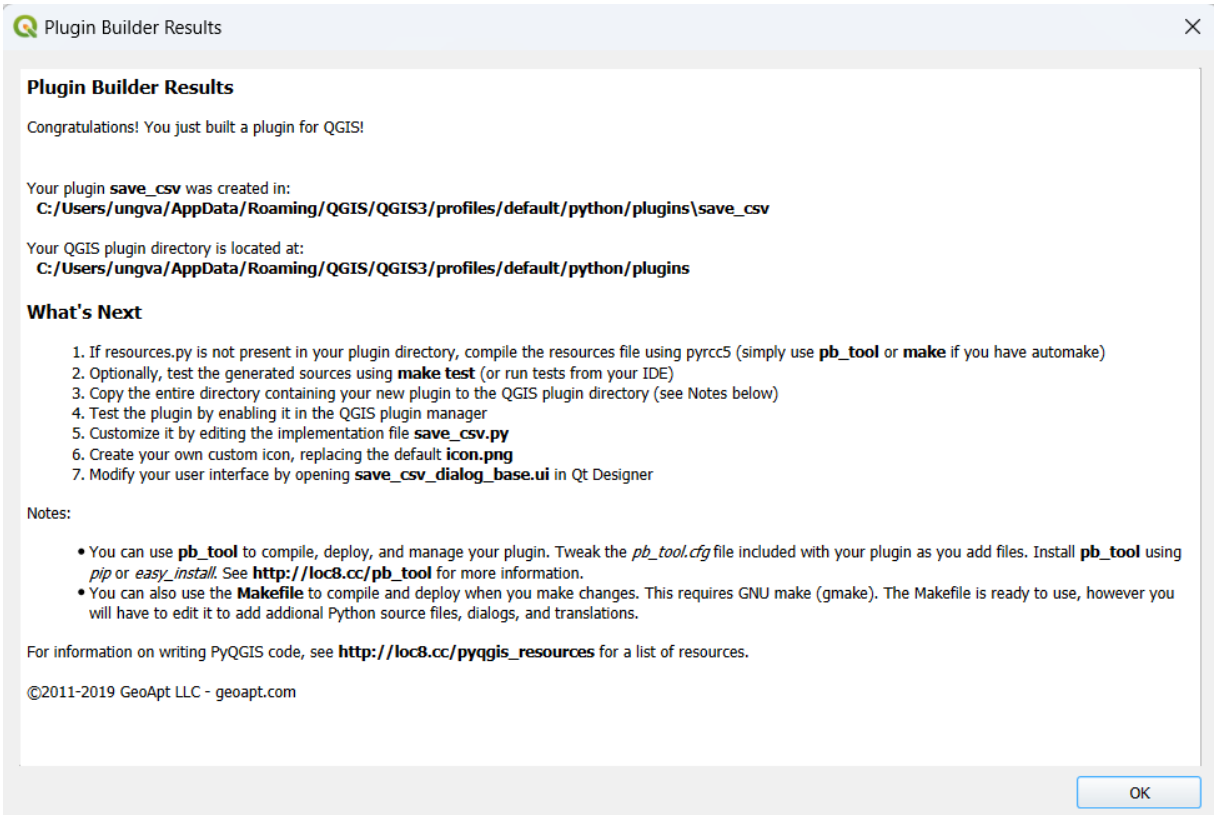
A *Flag the plugin as experimental* azt jelenti, hogy a modul a kísérleti modulok közé kerül majd be. Ebben az esetben ne felejtsük el bekapcsolni a *Manage and Install Plugin* menüben a *Settings*-nél a *Show also experimental plugin* lehetőséget, hogy lássuk az általunk létrehozott modult. A következő ablakban állítsuk be a mentés helyét. A legegyszerűbb a fejezet elején megadott a QGIS pluginek számára alapértelmezett helyre elmenteni a modult.



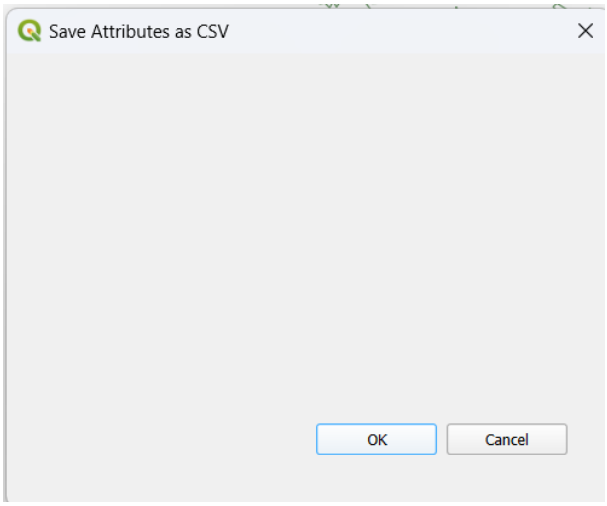
A Pluginek tárolására a QGIS a következő mappát használja, azzal a kitétellel, hogy meg kell adni a felhasználónevünket, és amennyiben a QGIS-ben profilokat használunk (vagyis nem a default-ot), adjuk meg a profil nevét, amelyik alatt látni szeretnénk a szkriptet.

C:/Users/SAJÁTFELHASZNÁLÓNÉV/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins

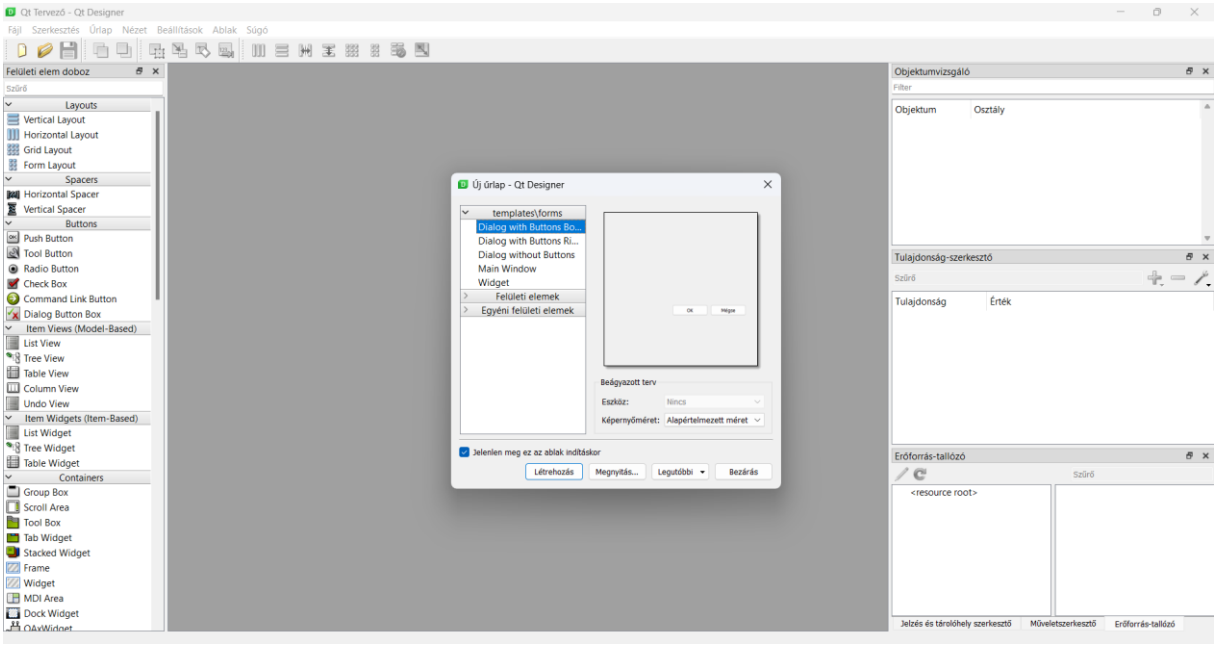




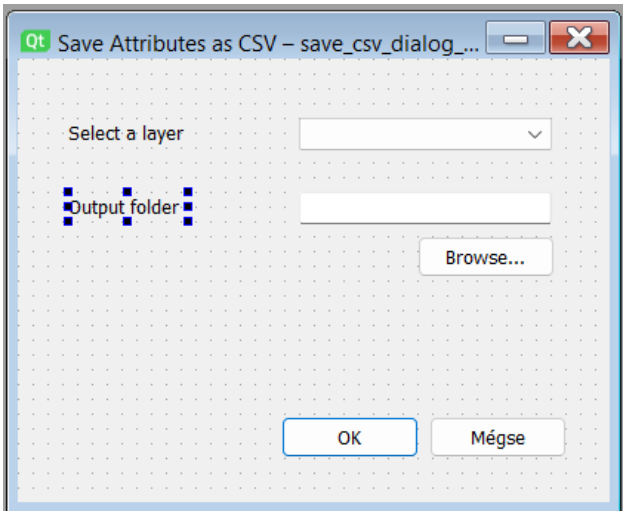
A fenti kép a modul sikeres létrehozását illusztrálja és a további teendőket is leírja. Első lépésként indítsuk újra a QGIS-t, menjünk bele a *Plugins* → *Manage and install plugins* menübe és kapcsoljuk be a plugint. Majd keressük a meg modult ott, amelyik menüt megjelöltük (én a plugin menübe tettem). Próbaként indítsuk el. Egyelőre egy OK és egy Mégsem gomb található rajta.



Ezután kezdjük el szerkeszteni a plugint. Készítsük el a felhasználói felületet a Qt Designer (with QGIS) egyszerűsített változatával. A program a QGIS-szel együtt települt, és a Start menüből indítható. Keressük meg a Megnyitás gombbal a plugin mappáját és abban a *save_csv_dialog_base.ui fájlt.* Ez a fájl tartalmazza a plugin felhasználói felületét. Ez egyelőre még nem csinál semmit.



Tegyünk fel a bal oldali eszközök közül a következőket: két címkét (*Label*), egy legördülő listát (*ComboBox*), egy szöveges beviteli mezőt (*Text edit*), és egy gombot (*Push Button*). A jobb oldali menüben szerkeszthető az eszközök nevek (*objectName*), illetve az értékük (a text pontban). Ne felejtsük elmenteni a változtatásokat.



Miután elkészült a felhasználói felület (*user interface*), nyissuk meg a Python fájlt, egy alkalmas kódszerkesztőben (QGIS Python konzol, IDLE, Notepad++, PyCharm stb.). A *save_csv.py* fájlt szerkesszük a következőképpen.

A fájl elején az importálandó moduloknál tegyük be a *QFileDialog*-ot (ez a dialógusablak megnyitásáért felel, amikor a mentés helyét választjuk ki), ez az objektum *a PyQt.QtWidgets* modulban található. Emellett importáljuk még a *core* osztályból a *QgsProject* eszközöket.

```
from qgis.PyQt.QtWidgets import QAction, QFileDialog
from qgis.core import QgsProject
```

```
"""
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication
from qgis.PyQt.QtGui import QIcon
from qgis.PyQt.QtWidgets import QAction, QFileDialog
from qgis.core import QgsProject
```

Keressük meg a *run(self)* függvényt, és elé szúrjuk be ezt a kódrészletet, ezzel elérhetjük, hogy beolvassa a rétegeket a legördülő listába (*comboBox*).

```
def select_output_file(self):
        filename, _filter = QFileDialog.getSaveFileName(self.dlg, "Select
output file ","", '*.csv')
        self.dlg.lineEdit.setText(filename)
```

```
def select_output_file(self):
    filename, _filter = QFileDialog.getSaveFileName(self.dlg, "Select   output file ","", '*.csv')
    self.dlg.lineEdit.setText(filename)

def unload(self):
    """Removes the plugin menu item and icon from QGIS GUI."""
```

Ezután a többi kiegészítés a *run(self)* függvény része lesz. Először írjuk meg azt a részt, amelyik meghívja az előbb definiált *select_output_file* függvényt, amikor a *Browse* gombra kattintok. Ennek hatására olvassuk be az összes réteget, amely a projektbe importálva van. Töröljük a legördülő lista eddigi tartalmát, majd töltsük fel az aktuális rétegekkel.

```
if self.first_start == True:
            self.first_start = False
            self.dlg = save_csvDialog()
            self.dlg.pushButton.clicked.connect(self.select_output_file)
        layers = QgsProject.instance().layerTreeRoot().children()
        self.dlg.comboBox.clear()
        self.dlg.comboBox.addItems([layer.name() for layer in layers])
```

Ha megvan a beállított fájlnév a szövegbeviteli mezőben, akkor olvassuk ki az elérési úttal együtt. Majd hozzuk létre a fájlt, nyissuk meg írásra. Ezután vegyük ki a legördülő lista épp beállított értékét (rétegnév), és keressük meg, hogy melyik réteg ez. Miután megvan a réteg, olvassuk ki a réteg mezőit.

```
if result:
      filename = self.dlg.lineEdit.text()
      with open(filename, 'w') as output_file:
              selectedLayerIndex = self.dlg.comboBox.currentIndex()
              selectedLayer = layers[selectedLayerIndex].layer()
              fieldnames = [field.name() for field in
selectedLayer.fields()]
```

Írjuk ki a mezőneveket a fájl fejlécébe. Utána indítsunk egy *for* ciklust, amelyben egyesével iterálunk végig az elemeken, és beírjuk a fájlba. A végén lezárjuk az írást. A beszúrt kép a kódok elhelyezését segíti.

```
              line = ','.join(name for name in fieldnames) + '\n'
              output_file.write(line)
              for f in selectedLayer.getFeatures():
                line = ','.join(str(f[name]) for name in fieldnames) + '\n'
                output_file.write(line)
```

```
    def run(self):
        """Run method that performs all the real work"""

        # Create the dialog with elements (after translation) and keep reference
        # Only create GUI ONCE in callback, so that it will only load when the plugin is started
        if self.first_start == True:
            self.first_start = False
            self.dlg = save_csvDialog()
            self.dlg.pushButton.clicked.connect(self.select_output_file)
        layers = QgsProject.instance().layerTreeRoot().children()
        # Clear the contents of the comboBox from previous runs
        self.dlg.comboBox.clear()
        # Populate the comboBox with names of all the loaded layers
        self.dlg.comboBox.addItems([layer.name() for layer in layers])

        # show the dialog
        self.dlg.show()
        # Run the dialog event loop
        result = self.dlg.exec_()
        # See if OK was pressed
        if result:
            filename = self.dlg.lineEdit.text()
            with open(filename, 'w') as output_file:
              selectedLayerIndex = self.dlg.comboBox.currentIndex()
              selectedLayer = layers[selectedLayerIndex].layer()
              fieldnames = [field.name() for field in selectedLayer.fields()]
              # write header
              line = ','.join(name for name in fieldnames) + '\n'
              output_file.write(line)
              # write feature attributes
              for f in selectedLayer.getFeatures():
                line = ','.join(str(f[name]) for name in fieldnames) + '\n'
                output_file.write(line)
```

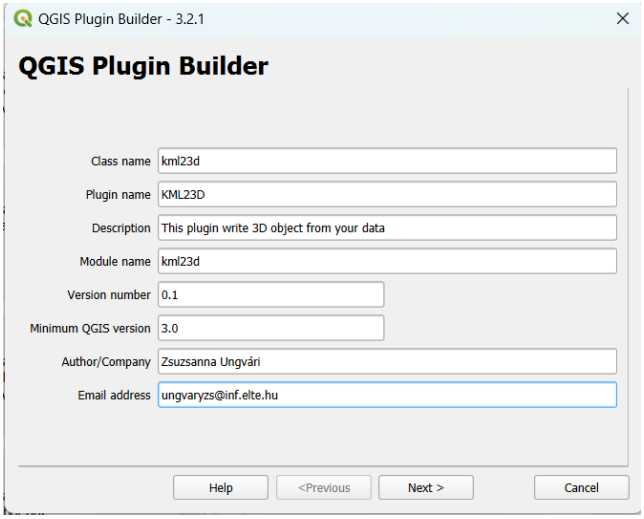
51. Egy szkript átalakítása pluginná

51_kml23d

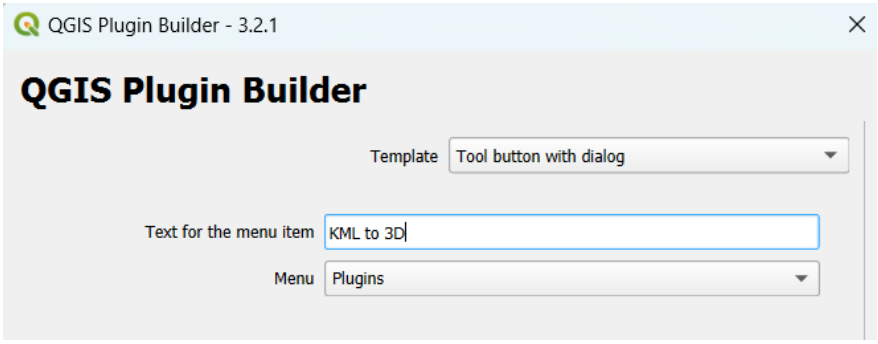
Alakítsuk át a 3D-s népsűrűség térképet generáló szkriptet pluginná! A feladat egyedi megoldását a 45. feladatban olvashatjuk, itt főleg az átalakítás lépéseire szeretnék kitérni. Arra

is figyelni kell a pluginná alakításkor, hogy ne csak arra az egy réteg, egyetlen oszlopára legyen működőképes a plugin, hanem általánosan, más fájlokon is működjön.

Az előző példában képes példákkal is illusztráltam a pluginkészítés első lépéseit. Ez a feladat is a *Plugin Builder* modulkészítő eszköz meghívásával kezdődik. Legyen a plugin neve *kml23d* (*KML to 3D*).

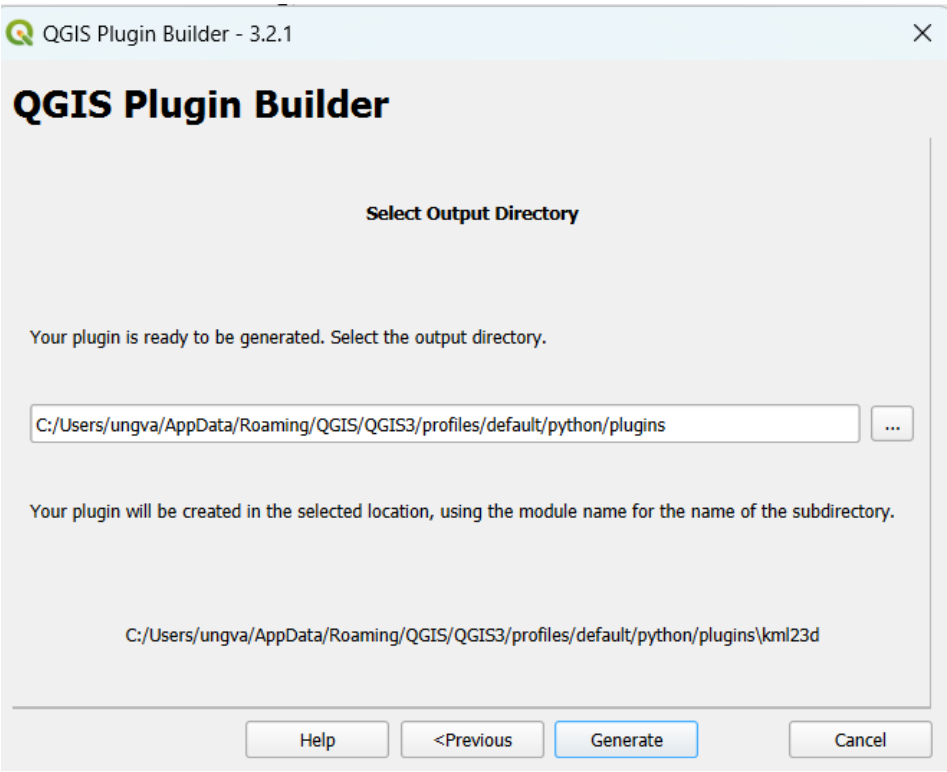


A következő lépésekben megadtam, hogy a *Plugin* menüben szeretném viszontlátni, és a *Tool button with dialog* sablont használom keretként.

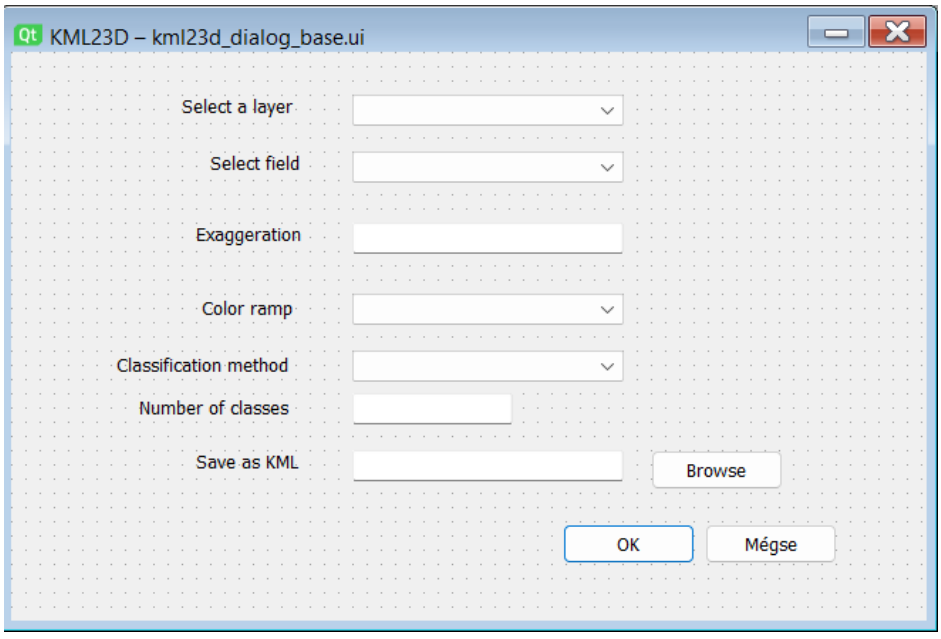


Most nem kérek hozzá segédfájlokat, egyébként hasznos lehet legalább a *pb_tool* generálása, ha a plugint megoszthatóvá szeretnénk tenni.

A pluginhoz létrejövő fájlokat rögtön a QGIS erre a célra létrehozott, alapértelmezett mappájába készítem el.



Ezután megkezdem a sablon felhasználói felületének szerkesztését *Qt Designer*ben.



Szükség lesz négy legördülő listára (*comboBox*), három szöveges beviteli mezőre (*lineEdit*) és egy gombra (*pushButton*). Az OK és a Mégse gombok adottak. Emellett adjunk hozzá címkéket is, amelyek a sorok magyarázatát adják meg.

A plugin részletes működése

Az első listában kiválaszthatja a felhasználó, hogy melyik réteget szeretné használni. A második listában a kiválasztott rétegen található mezőket olvashatjuk. Ebből kell beállítani azt, amelyikből 3D-s tartalmat szeretnénk generálni, ennek szám adattípusnak kell lennie.

Ezután adjuk meg a túlmagasítást, a használt színskálát, az osztályozás módszerét, és az osztályok számát. Végül írjuk be a mentés helyét. Ezután az OK gombra kattintva lefut a

program, és legenerálja a 3D-s KML fájlt. Ezt megnyitni például a *Google Earth Pro*-ben tudjuk legegyszerűbben.

A kód megvalósítása

A fentiek alapján a *QtDesigner*ben átneveztem az ablakra feltett eszközöket. A rétegválasztó a *layer0,* a mezőválasztó a *field0*, a túlmagasítást beállító szövegbeviteli mező az *exagg0,* a színskálát beállító lista a *colorRamp0*, az osztályozási módszert választó lista a *classMethod0,* az osztályok számát megadó mező pedig a *classes0* nevet kapta. Végül pedig a mentési nevet megadó szöveges mezőt *save0*-nak és a gombot pedig *browse0*-nak hívom.

Ezután a QGIS Python konzolban szerkesztésre megnyitottam a *kml23d.py* fájlt. A feladat kivitelezéséhez szükség lesz a qgis core osztály függvényeire (pl. vetületi transzformáció) és az interfészre is. Ezeket beimportálom.

```
from qgis.utils import iface
from qgis.core import *
```

Az előző CSV-s feladatban szükség volt szöveges fájl mentésére. Itt is ez a helyzet, ehhez ugyanazokat a kódokat fogom használni a kiterjesztés módosításával, és az eszközök nevének frissítésével.

```
def select_output_file(self):
        filename, _filter = QFileDialog.getSaveFileName(self.dlg, "Select
output file ","", '*.kml')
        self.dlg.save0.setText(filename)
```

A *run()* függvényben is megadom, hogyha a gombra kattintunk, hívja meg a *select_output_file()* függvényt.

```
def run(self):
      if self.first_start == True:
            self.first_start = False
            self.dlg = kml23dDialog()
            self.dlg.browse0.clicked.connect(self.select_output_file)
```

A következő részben minden kód a *run()* függvény része. Először is töltsük a fel a rétegek nevével a *layer0* listát, és a *field0* listát pedig a kiválasztott mezők neveivel. Ha réteget váltunk törlődjön a lista tartalma és újra adjuk hozzá a mezőket (aktualizáljuk a legördülő lista megjelenő elemeit). A *QgsProject.instance().layerTreeRoot().children()* kiveszi a rétegbe betöltött elemeket. A *clear()* függvény üríti a listák eddigi tartalmát. Majd nézzük meg mi a kiválasztott réteg indexe (a plugin betöltésénél ez a legfelső réteg), majd töltsük fel a réteg mezőivel a másik listát. Kell egy függvény is ugyanezekkel a parancsokkal, amelyet akkor hívunk meg, ha a *layer0* legördülő lista tartalma megváltozik, ez lesz a *field_select()* függvény.

```
      layers = QgsProject.instance().layerTreeRoot().children()
      self.dlg.layer0.clear()
      self.dlg.field0.clear()
      self.dlg.layer0.addItems([layer.name() for layer in layers])
      selectedLayerIndex = self.dlg.layer0.currentIndex()
      selectedLayer = layers[selectedLayerIndex].layer()
      fields = [field.name() for field in selectedLayer.fields()]

      def field_select():
            self.dlg.field0.clear()
            selectedLayerIndex = self.dlg.layer0.currentIndex()
            selectedLayer = layers[selectedLayerIndex].layer()
```

```
            fields = [field.name() for field in selectedLayer.fields()]
            self.dlg.field0.addItems(fields)
```

Van még két lista, ezek az osztályozási módszert, illetve a színskálát adják meg. Töltsük fel őket adattal. Az osztályozási módszernél a módszer nevét adom meg (most választható lesz a természetes törések, az egyenlő intervallumok módszere, és az egyenlő darabszámok módszere), a színskáláknál néhány QGIS-ben alapértelmezetten létező színskála nevét.

```
    self.dlg.layer0.currentIndexChanged.connect(field_select)
    methodList=['Jenks','Equal Interval','Quantile']
    self.dlg.classMethod0.addItems(methodList)
    colorList=['Reds','Blues','Greens','Spectral']
    self.dlg.colorRamp0.addItems(colorList)
```

Az összes többi kód pedig az OK gomb megnyomása után fog végrehajtódni az *if result* elágazásban. Először is olvasssuk ki mentés helyének teljes elérési útját, a túlmagasítás értékét, az osztályok számát, az osztályozás módszerét. Mivel az utóbbinál csak nevet adtunk meg a listában, most az elágazásban a névhez a *QgsClassification* metódussal kell hozzárendelni az osztályozás módszerét. Végül a megadott színskálák közül olvassuk be a lista értékét a *currentText()* függvénnyel.

A korábban tanultak alapján (12. feladat) hozzunk létre egy új stílust a színskálához.

```
    if result:
        filename = self.dlg.save0.text()
        exagg = int(self.dlg.exagg0.text())
        num_classes = int(self.dlg.classes0.text())
        classMethod=self.dlg.classMethod0.currentText()
        if classMethod == 'Jenks':
            classification_method=QgsClassificationJenks()
        elif classMethod == 'Equal Interval':
            classification_method=QgsClassificationEqualInterval()
        else:
            classification_method=QgsClassificationQuantile()
        ramp_name=self.dlg.colorRamp0.currentText()
        default_style = QgsStyle().defaultStyle()
        color_ramp = default_style.colorRamp(ramp_name)
```

A következő lépésben kezdjük el létrehozni és szerkeszteni a KML fájlt (*with open*), mint egy szöveges fájl. Kiolvassuk a kiválasztott réteg és a mező nevét a legördülő listából. A legegyszerűbb talán úgy meghívni az osztályozást, ha magát a réteget is kiszínezem a beállítások alapján. Ehhez a *QgsGraduatedSymbolRenderer()*-t használom a 12. feladatban megismert módon.

```
        with open(filename, 'w') as f:
              selectedLayerIndex = self.dlg.layer0.currentIndex()
              layer = layers[selectedLayerIndex].layer()
              fieldnames = [field.name() for field in layer.fields()]

              renderer = QgsGraduatedSymbolRenderer()
              renderer.setClassAttribute(self.dlg.field0.currentText())
              renderer.setClassificationMethod(classification_method)
              renderer.updateClasses(layer, num_classes)
              renderer.updateColorRamp(color_ramp)
```

```
                layer.setRenderer(renderer)

                layer.triggerRepaint()
                layer_tree=iface.layerTreeView()
                layer_tree.refreshLayerSymbology(layer.id())
```

Megírjuk a KML fájl fejlécét, ahova a stílusokat is bele kell tennünk. Most a plugin beállításai szerint kell eljárnunk. Vagyis egy *for* ciklusban olvassuk ki az előbbi lépésben beállított osztályokat, illetve azok színeit. Mivel a KML nem a hagyományos értelemben vett hexadecimális kódokat használja, kisebb módosítás szükséges. A „hagyományos”, tehát a QGIS vagy pl. am CSS-ben használt hexadecimális kódok így néznek ki: *#FF0000.* Ez a kód a vörös szín. A kettős kereszt után az RRGGBB sorrendben értékeket 0–9 és A–F betűig jelöljük, minden színhez (R, G, B) két betű tartozik. Tehát ennél a színnél a vörösből maximális érték az FF, a zöldből és kékből a minimális érték 00. Minden RGB szín ezzel a módszerrel könnyen megadható. A KML hexadecimális értékeiben annyi változtatás történik, hogy az átlátszóságot is rögzítjük (plusz két szám vagy betű), eltűnik a ’#’, valamint a sorrend is megfordul! Tehát AABBGGRR, vagyis átlátszóság, kék, zöld és vörös színértékek a sorrend, ugyanúgy 0–9-cel és A–F-el jelölve. Így a vörösnek a kódja ff0000ff lesz. Ezt a hagyományos hexa kódokból könnyen elkészíthető a sorrend felcserélésével és az átlátszóság hozzáadásával.

```
                f.write('<?xml version="1.0" encoding="UTF-8"?><kml
xmlns="http://www.opengis.net/kml/2.2"
xmlns:gx="http://www.google.com/kml/ext/2.2"
xmlns:kml="http://www.opengis.net/kml/2.2" >')
                f.write('<Document><name>Nepsuruseg.kml</name>
<open>1</open><Folder>')
                count=0
                for i in renderer.legendSymbolItems():
                    count=count+1
                    color=i.symbol().color().name()
                    szin='ff'+color[5]+color[6]+color[3]
+color[4]+color[1]+color[2]
                    f.write('<Style id="'+str(count)+'"><LineStyle>
<color>ff000000</color></LineStyle><PolyStyle><color>'
+str(szin)+'</color></PolyStyle></Style>')
```

Végrehajtjuk a vetületi transzformációt, vagyis EOV-ből földrajzi koordinátákat képezünk.

```
                source_crs =
QgsCoordinateReferenceSystem(layer.crs().authid())
                target_crs = QgsCoordinateReferenceSystem("EPSG:4326")
```

Utána egyesével végigmegyünk az elemeken, létrehozzuk a *Placemark*ot. Ez a lépés elég hasonló 45. feladathoz, így csak a szkript elejét ismertetem, ahol változás van. Mivel az egyes osztályok értékhatárai attól függnek, hogy éppen melyik osztályozási módszert és hány csoportot képezünk, ezért a *renderer.ranges()* függvénnyel ki kell olvasni az egyes osztályokat, és meg kell határozni, hogy az aktuális elem melyikbe tartozik. Ez alapján írom meg a *styleUrl*-t. Emellett a poligon magasságot a túlmagasítás mező értékével szorzom meg.

```
                for feature in layer.getFeatures():
                    f.write('\n <Placemark>')
                    f.write('<name>'+str(feature.attributes()[1])+
'</name>')
                    count=0
                    for i in renderer.ranges():
                        count=count+1
```

```
                        if feature[self.dlg.field0.currentText()]>=i[0] and
feature[self.dlg.field0.currentText()]<=i[1]:
                            f.write('<styleUrl>#'+str(count)+'</styleUrl>')

                    mag=feature[self.dlg.field0.currentText()]*exagg
                    f.write('<Polygon><tessellate>1</tessellate>
<altitudeMode>relativeToGround</altitudeMode>
<extrude>1</extrude><outerBoundaryIs><LinearRing><coordinates>')
                    geom = feature.geometry()
                    geom.transform(QgsCoordinateTransform(source_crs,
target_crs, QgsProject.instance()))
```

A feladat további része megegyezik a 45.-ös feladattal.

Továbbfejlesztési lehetőségek

- Tegyünk fel egy újabb *comboBox*-ot, amelyben meghatározhatjuk, hogy a poligon nevét honnan szedje ki a <*name*> tagbe.

- Adjunk hozzá további osztályozókat és színskálákat.

- Készítsünk használati útmutatót a pluginhez.

- Kezeljük le azokat a hibákat, amelyek abból adódnak, hogy a felhasználó nem töltötte ki valamelyik, pl. szöveges mezőt. Ezt megvalósíthatjuk pl. egy *default* érték megadásával, vagy egy figyelemfelkeltő üzenettel is (*messageBar*). Vizsgáljuk meg a szöveges mezőkben megfelelő típusú adatot adott-e meg a felhasználó.

- Adjunk hozzá a fájlírási folyamat állapotát jelző üzenetet (*messageBar*-t), ez praktikus lehet nagy fájlok esetén.

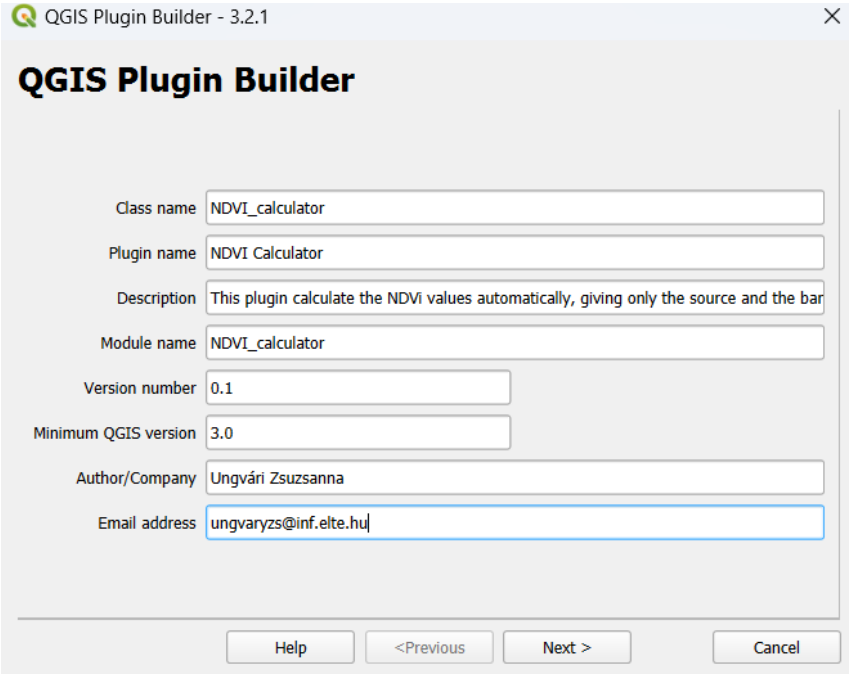
52. Egy Processing plugin készítése

Használjuk fel a 43. feladatot a Processing plugin elkészítéséhez. A feladatban készítsünk egy olyan plugint, amely kiszámítja az NDVI értéket. Ehhez bemeneti adatként egy többcsatornás műholdképet kell megadni, valamint a csatornák sorszámát, amelyek tartalmazzák a közeli infravörös és a vörös értékeket.

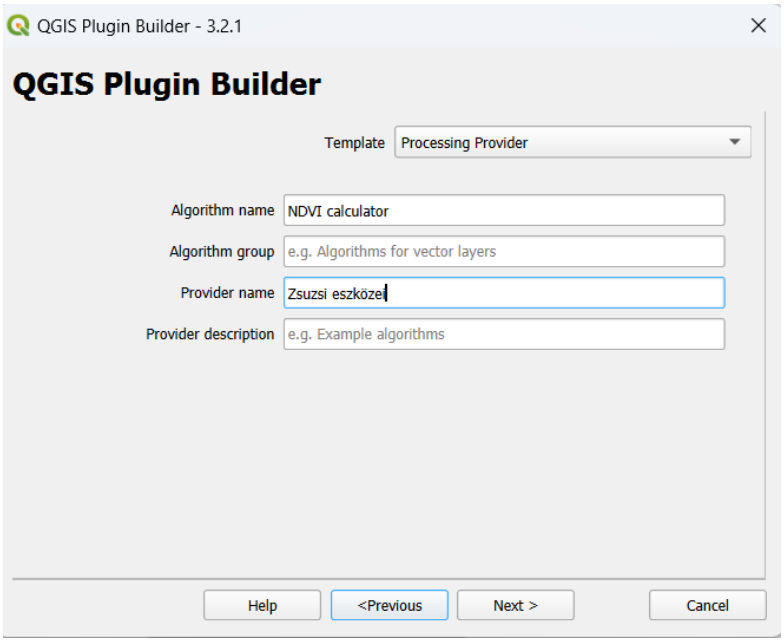
Mi az a Processing plugin?

A Processing pluginek abban különböznek a hagyományos pluginektől, hogy itt nem kell megterveznünk a grafikus interfészt, nem készül UI kiterjesztésű fájl. Emellett automatikusan a Processing menübe kerülnek, onnan kell elindítani őket.

Első lépésként a kapcsolódó fájlokat és függvényeket kell legyártani, ezeket legegyszerűbben a *Plugin Builder*rel tudjuk megtenni. A beállítások gyakorlatilag egy helyen térnek el az előzőektől. Az első lépésben megadom a plugin nevét, és az osztály nevét.

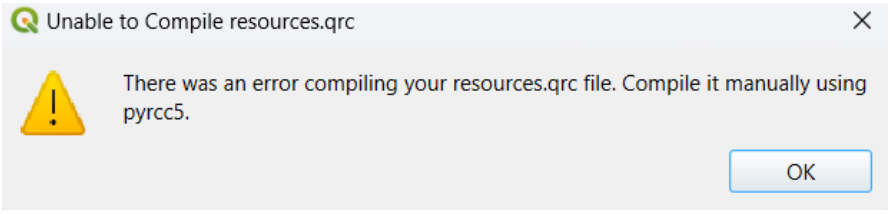


Amikor a plugin generáló alkalmazás megkérdezi, hogy milyen vázlatot (*templatet*) szeretnék használni, akkor a *Processing Provider*t használjuk, és adjuk meg az algoritmus nevét és a *Provider* nevét.



A mentés helye megegyezik a hagyományos pluginokéval.

Szeretném megjegyezni, hogy az általam használt verzió nem tudta tökéletesen legenerálni a *Processing plugin*hez tartozó összes fájlt. A *resorurces.qrc* és a *resources.py* nem készül el. Viszont ha a *Manage and install Plugins* menüben szeretnénk aktiválni a plugint, szükség lesz erre a két fájlra. Áthidaló megoldás lehet, ha egy normál (bármely plugin) mappájából átmásoljuk ezt a két fájl. Majd a QRC kiterjesztésű fájlt megnyitjuk és átírjuk a mappanevét a saját mappanevünkre.



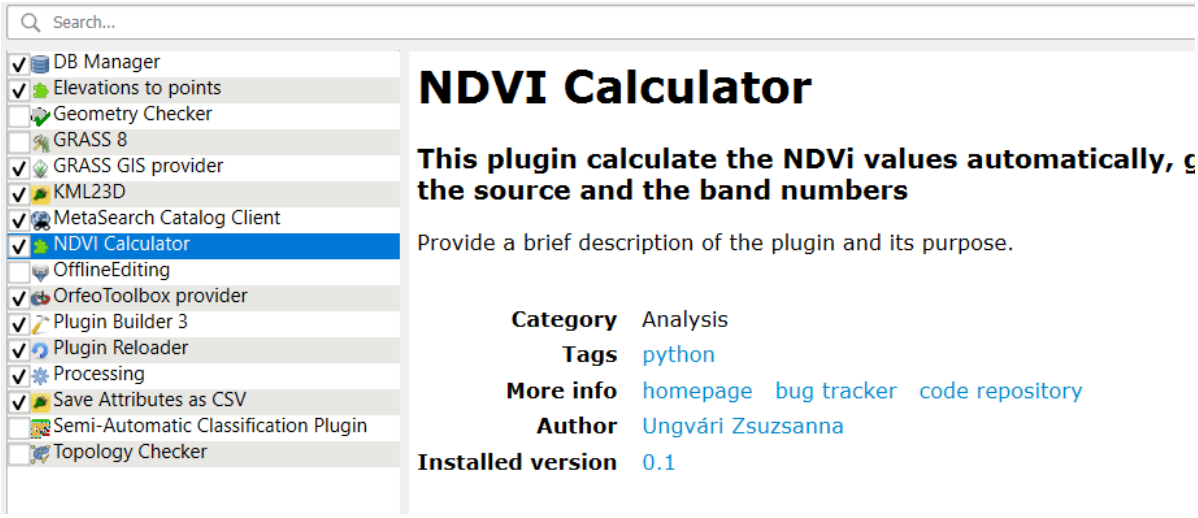
A kiemelt részt javítsuk át

```
<RCC>
    <qresource prefix="/plugins/elev_to_points" >
        <file>icon.png</file>
    </qresource>
</RCC>
```

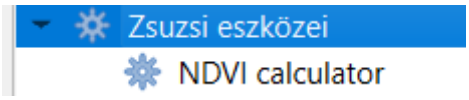
erre:

```
<RCC>
    <qresource prefix="/plugins/NDVI_calculator" >
        <file>icon.png</file>
    </qresource>
</RCC>
```

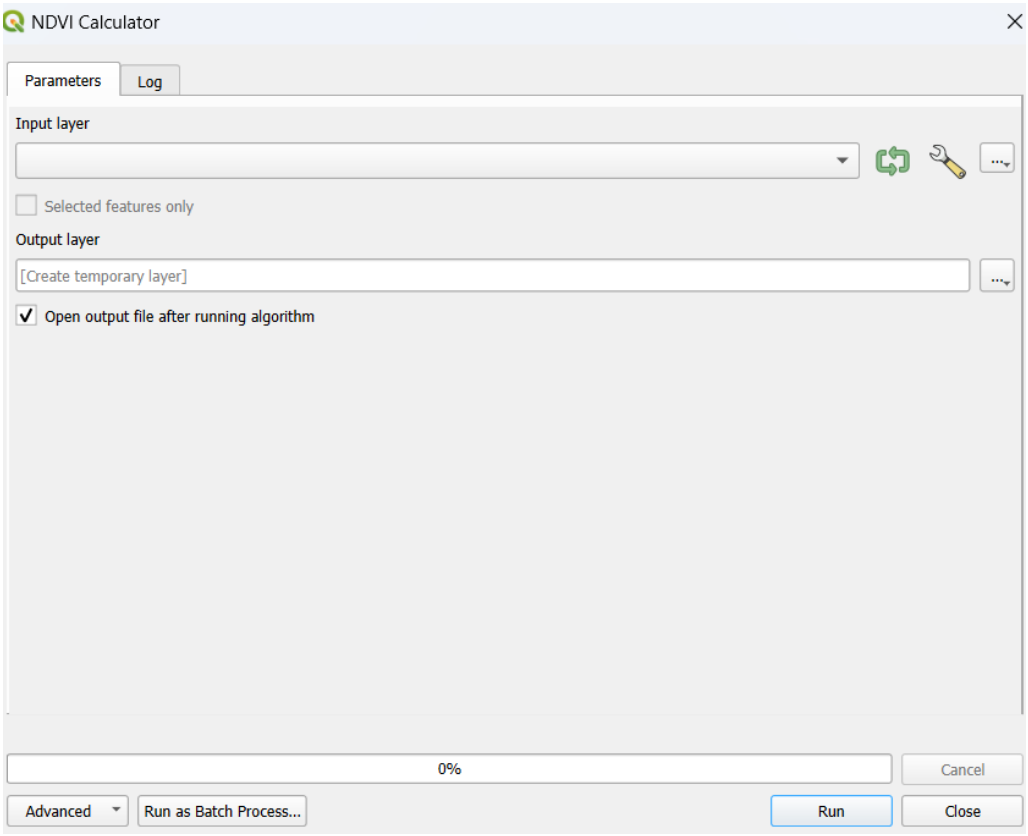
Ezután indítsuk újra a QGIS-t és aktiváljuk a *Manage and install plugin* menüben az *NDVI calculator*t.



Az elkészült eszközt keressük a *Processing* menüben az általunk megadott néven, és indítsuk el.



Indítás után ez fogad minket. Minden *Processing* pluginhez felkerül egy vektoros adatok beolvasására képes mező (*input layer*), és egy kimeneti (*output*) adatok mentési helyét megadó legördülő lista.



Ezeket kellene módosítani. Olvassunk be egy raszteres réteget, és két szöveges mezőt (ezeket még hozzá kell adnunk). Ehhez nyissuk meg *NDVI_calculator_algorithm.py* fájlt. Ez fogja tartalmazni az algoritmust, ide tanácsos helyezni a további futáshoz szükséges kódjainkat. A fájlok számos kommentet tartalmaznak, amelyek segítik a *Processing plugin*ek működésének értelmezését.

Nézzük át a fájl alapvető függvényeit, mielőtt nekiállínánk a szerkesztésnek. Ebben a fájlban egy osztály található azon a néven, amit megadtunk, esetünkben ez a *NDVI_calculatorAlgorithm(QgsProcessingAlgorithm)*. Az osztályban a következő függvények vannak definiálva: *initAlgorithm(self, config)* – ebben adjuk meg a be és kimeneti adatokat, def *processAlgorithm(self, parameters, context, feedback)* – ebben helyezzük el a feldolgozáshoz szükséges kódot, ezen kívül a *name(self)* és *displayName(self)* függvények felelnek az plugin nevének megjelenítéséért, *group(self)* és a *groupId(self)* a Processing menüben jelenítik meg az eszközcsoportot, valamint a *tr(self, string)* és *a createInstance(self)* segít példányosítani az az eszközt.

Nekünk tehát az első kettő függvénnyel kell foglalkoznunk, illetve természetesen adhatunk hozzá mi is további függvényeket a fájlhoz (bár esetünkben ez nem szükséges).

Első lépésként cseréljük le a vektoros input mezőt egy raszteresre és tegyünk utána két szöveges beviteli sávot. Ezt a részt töröljük:

```
self.addParameter(
            QgsProcessingParameterFeatureSource(
                self.INPUT,
                self.tr('Input layer'),
                [QgsProcessing.TypeVectorAnyGeometry]
            )
        )
```

Helyette tegyük be ezt:

```
self.addParameter(
            QgsProcessingParameterRasterLayer (
                self.INPUT,
                self.tr('Input raster layer'),
                [QgsProcessing.TypeRaster]
            )
        )
        self.addParameter(
            QgsProcessingParameterString(
                self.INPUTNIR,
                self.tr('Band number for NIR')

            )
        )
        self.addParameter(
           QgsProcessingParameterString(
                self.INPUTRED,
                self.tr('Band number for Red')

            )
        )
```

A raszteres rétegeket beolvasó legördülő listát létrehozó függvény neve *QgsProcessingParameterRasterLayer()*. Az első paramétere az input változó, a második paraméter a szövegmező tartalma a lista felett, a harmadik paraméterrel tudjuk leszűrni a raszteres rétegeket (*QgsProcessing.TypeRaster*).

A függvény előtt az osztályban ne felejtsük el definiálni a kövegkező változókat, hogy ezek a függvények használhassák őket.

```
OUTPUT = 'OUTPUT'
INPUT= 'INPUT'
INPUTRED='INPUTRED'
INPUTNIR='INPUTNIR'
```

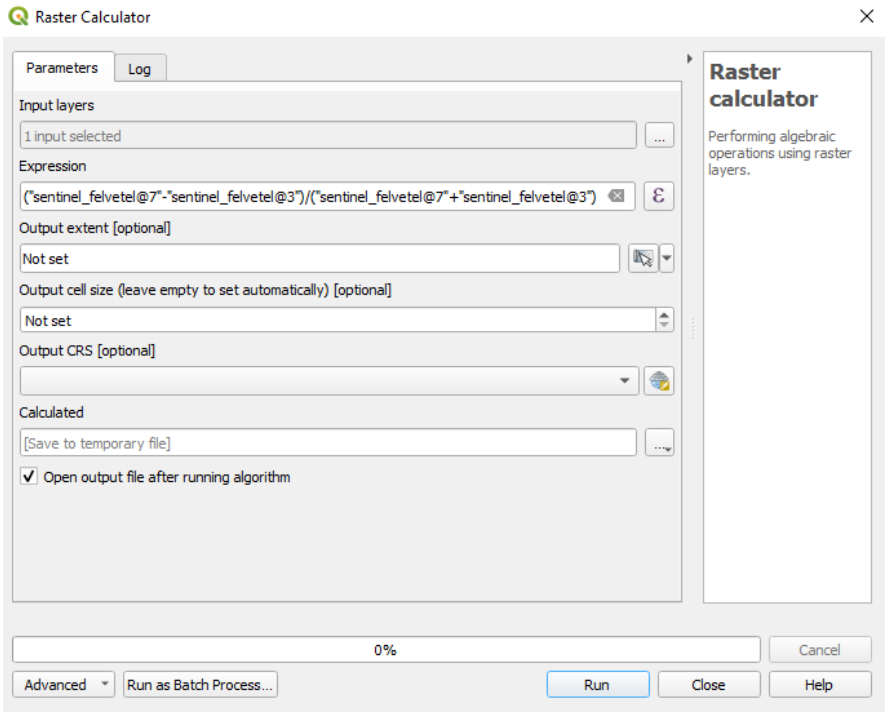
A szöveges legördülő lista defininálása hasonlóan történik a *QgsProcessingParameterString()* függvénnyel, amely első paramétere a beviteli mező, a második annak felirata. Ha megagyunk, rátérhetünk a kimenetre.

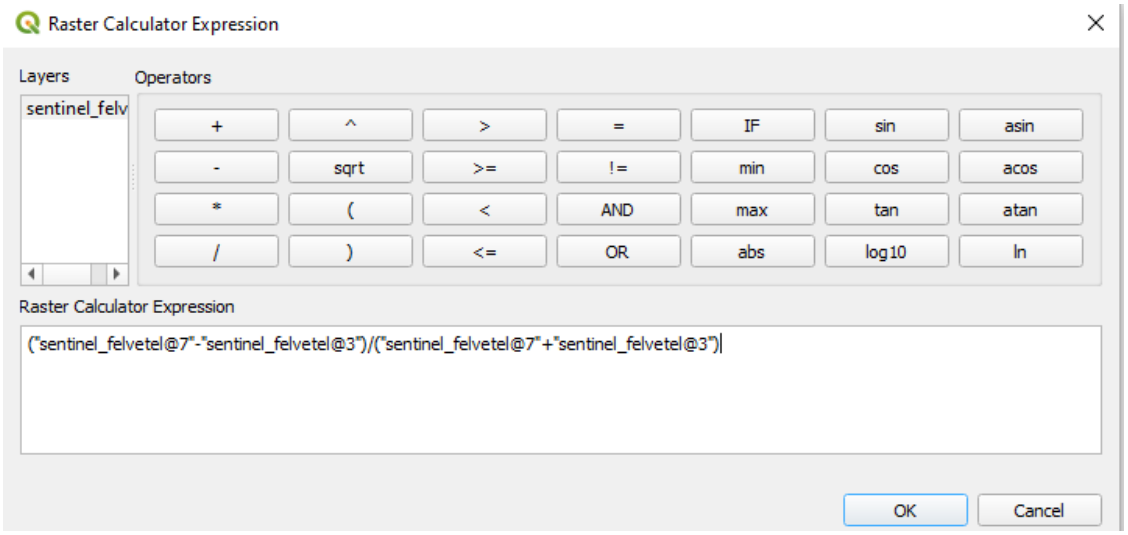
A kimenet raszteres réteg lesz, ezért cseréljük itt a *QgsProcessingParameterFeatureSink()* függvényt ennek raszteres változatára, a *QgsProcessingParameterRasterDestination()*-ra.

```
self.addParameter(
            QgsProcessingParameterRasterDestination(
                self.OUTPUT,
                self.tr('Output NDVI layer')
            )
        )
```

A *processAlgorithm()* függvényhez adjuk hozzá azt a kódot, amellyel kiszámítjuk az NDVI értékét. Ehhez felhasználok már egy meglévő *Processing* eszközt, a *Raster Calculator*t. A szükséges kód előállításához nyissuk meg a beépített – natív – QGIS eszközök közül ezt a menüt, töltsük ki megfelelően az egyes mezőket a képlettel együtt, majd másoljuk ki ez alapján a Python kódot. Ezt kissé módosítottam, a paraméterezését külön objektumban helyeztem el. A kép csatornáinak sorszámát a kép szöveges mezőből olvasom be, értelemszerűen csak olyan

felvételeknél fog működni, amelyeknél egy képben van elmentve a közeli infravörös és vörös sáv.






```
def processAlgorithm(self, parameters, context, feedback):
        raster=self.parameterAsRasterLayer(parameters,self.INPUT, context)
        nir=self.parameterAsString(parameters,self.INPUTRED, context)
        red=self.parameterAsString(parameters,self.INPUTNIR, context)
        exp='("sentinel_felvetel@'+nir+'"-
"sentinel_felvetel@'+red+'")/("sentinel_felvetel@'+nir+'"+"sentinel_felvete
l@'+red+'")'
        params = {
        'LAYERS' : raster,
        'EXPRESSION':exp,
        'EXTENT': None,
        'CRS': None,
        'OUTPUT' :
self.parameterAsOutputLayer(parameters,self.OUTPUT,context)
```

```
        }
        res =processing.run('native:rastercalc', params)
        return {self.OUTPUT: res}
```

Miután elkészültünk, teszteljük le az új Plugin működését.

Lehetséges továbbfejlesztési ötlet

A plugin kiegészíthető hibakezelő formulákkal, pl. nézzük meg, hogy valóban számokat írtunk-e be a beviteli mezőkbe és azok értéke egyezik-e a csatornák számával, létezik-e a megadott csatorna.