
PyQGIS developer cookbook

Release 2.8

QGIS Project

July 30, 2016

1	Introducere	1
1.1	Rularea codului Python atunci când pornește QGIS	1
1.2	Consola Python	2
1.3	Plugin-uri Python	2
1.4	Aplicaii Python	3
2	Încărcarea proiectelor	5
3	Încărcarea Straturilor	7
3.1	Straturile Vectoriale	7
3.2	Straturile Raster	8
3.3	Registrul Straturilor de Hartă	9
4	Utilizarea straturilor raster	11
4.1	Detaliile stratului	11
4.2	Drawing Style	11
4.3	Recitirea straturilor	13
4.4	Interogarea valorilor	13
5	Utilizarea straturilor vectoriale	15
5.1	Retrieving informations about attributes	15
5.2	Selectarea entităților	15
5.3	Iterații în straturile vectoriale	15
5.4	Modificarea straturilor vectoriale	17
5.5	Modificarea straturi vectoriale prin editarea unui tampon de memorie	19
5.6	Crearea unui index spaial	19
5.7	Scrierea straturilor vectoriale	20
5.8	Furnizorul de memorie	21
5.9	Aspectul (simbologia) straturilor vectoriale	22
5.10	Lecturi suplimentare	29
6	Manipularea geometriei	31
6.1	Construirea geometriei	31
6.2	Accesarea geometriei	31
6.3	Predicate i operaiuni geometrice	32
7	Proiecii suportate	35
7.1	Sisteme de coordonate de referină	35
7.2	Proiecii	36
8	Folosirea suportului de hartă	37
8.1	Încapsularea suportului de hartă	37
8.2	Folosirea instrumentelor în suportul de hartă	38

8.3	Benzile elastice i marcajele nodurilor	39
8.4	Dezvoltarea instrumentelor personalizate pentru suportul de hartă	40
8.5	Dezvoltarea elementelor personalizate pentru suportul de hartă	41
9	Randarea hărților i imprimarea	43
9.1	Randarea simplă	43
9.2	Randarea straturilor cu diferite CRS-uri	44
9.3	Generarea folosind Compozitorul de hări	44
10	Expresii, filtrarea i calculul valorilor	47
10.1	Parsarea expresiilor	48
10.2	Evaluarea expresiilor	48
10.3	Exemple	48
11	Citirea i stocarea setărilor	51
12	Comunicarea cu utilizatorul	53
12.1	Afiarea mesajelor. Clasa QgsMessageBar	53
12.2	Afiarea progresului	54
12.3	Jurnalizare	55
13	Dezvoltarea plugin-urilor Python	57
13.1	Scrierea unui plugin	57
13.2	Coninutul Plugin-ului	58
13.3	Documentaie	62
14	Setările IDE pentru scrierea i depanarea de plugin-uri	63
14.1	O notă privind configurarea IDE-ului în Windows	63
14.2	Depanare cu ajutorul Eclipse i PyDev	64
14.3	Depanarea cu ajutorul PDB	68
15	Utilizarea straturilor plugin-ului	69
15.1	Subclasarea QgsPluginLayer	69
16	Compatibilitatea cu versiunile QGIS anterioare	71
16.1	Meniul plugin-ului	71
17	Lansarea plugin-ului dvs.	73
17.1	Metadate i nume	73
17.2	Codul i ajutorul	73
17.3	Depozitul oficial al plugin-urilor python	74
18	Fragmente de cod	77
18.1	Cum să apelăm o metodă printr-o combinaie rapidă de taste	77
18.2	Inversarea Stării Straturilor	77
18.3	Cum să accesezi tabelul de atribute al entităților selectate	77
19	Biblioteca de analiză a reelelor	79
19.1	Informaii generale	79
19.2	Construirea unui graf	79
19.3	Analiza grafului	81
	Index	87

Introducere

Acest document are rolul de tutorial dar i de ghid de referină. Chiar dacă nu prezintă toate cazurile de utilizare posibile, ar trebui să ofere o bună imagine de ansamblu a funcționalităților principale.

Începând cu versiunea 0.9, QGIS are suport de scriptare opional, cu ajutorul limbajului Python. Ne-am decis pentru Python deoarece este unul dintre limbajele preferate în scriptare. PyQGIS depinde de SIP i PyQt4. S-a preferat utilizarea SIP în loc de SWIG deoarece întregul cod QGIS depinde de bibliotecile Qt. Legarea Python de Qt (PyQt) se face, de asemenea, cu ajutorul SIP, acest lucru permiând integrarea perfectă a PyQGIS cu PyQt.

TODO: Getting PyQGIS to work (Manual compilation, Troubleshooting)

There are several ways how to use QGIS python bindings, they are covered in detail in the following sections:

- rularea automată a codului Python atunci când pornește QGIS
- scrierea comenzilor în consola Python din QGIS
- crearea în Python a plugin-urilor
- crearea aplicațiilor personalizate bazate pe QGIS API

Există o referină [API QGIS completă](#) care documentează clasele din bibliotecile QGIS. API-ul QGIS pentru Python este aproape similar cu cel pentru C++.

There are some resources about programming with PyQGIS on [QGIS blog](#). See [QGIS tutorial ported to Python](#) for some examples of simple 3rd party apps. A good resource when dealing with plugins is to download some plugins from [plugin repository](#) and examine their code. Also, the `python/plugins/` folder in your QGIS installation contains some plugin that you can use to learn how to develop such plugin and how to perform some of the most common tasks

1.1 Rularea codului Python atunci când pornește QGIS

Există două metode distincte de a rula cod Python de fiecare dată când pornește QGIS.

1.1.1 Variabila de mediu PYQGIS_STARTUP

Puteți rula cod Python mai înainte de finalizarea inițializării QGIS, indicând în variabila de mediu PYQGIS_STARTUP calea spre un fișier Python existent.

Această metodă este rar utilizată, dar merită menționată, deoarece reprezintă una din metodele de a rula cod Python în QGIS, și pentru că acest cod se va executa înainte de finalizarea inițializării QGIS. Această metodă este foarte utilă pentru curățarea `sys.path`, care poate conține căi nedorite, sau pentru izolarea/încărcarea căilor mediului inițial fără a fi necesară instalarea unui mediu virtual, cum ar fi homebrew sau MacPorts pe Mac.

1.1.2 Fișierul `startup.py`

De fiecare dată când pornește QGIS, directorul home al utilizatorului Python(de obicei `.qgis2/python`) este căutat un fișier numit `startup.py`, dacă acesta există, este executat de către interpretorul Python integrat.

1.2 Consola Python

Pentru scripting, se poate utiliza consola Python integrată. Aceasta poate fi deschisă din meniul: *Plugins* → *Consola Python*. Consola se deschide ca o fereastră utilitară, non-modală:

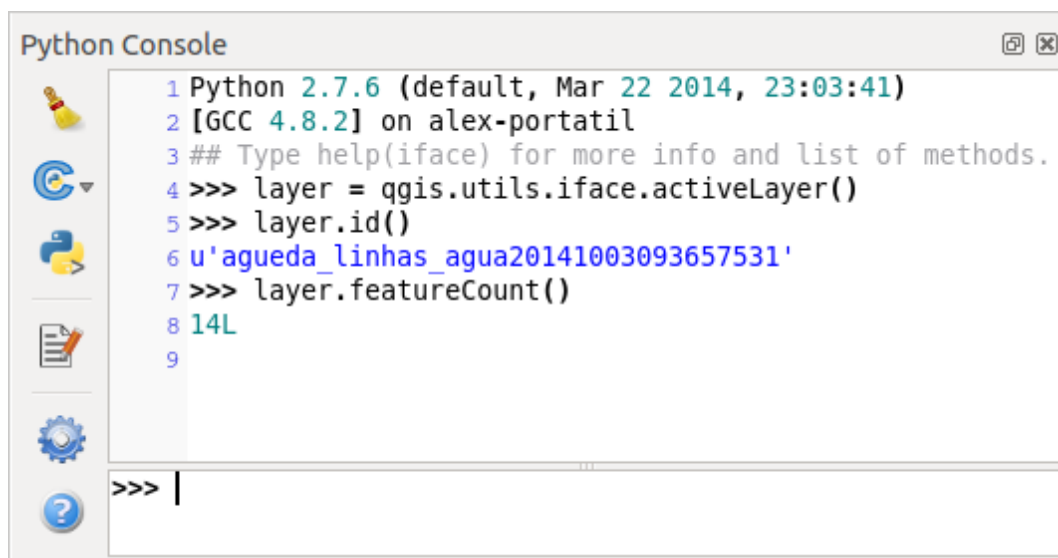


Figure 1.1: Consola Python din QGIS

Captura de ecran de mai sus ilustrează cum să puteți obține accesul la stratul curent selectat în lista straturilor, pentru a-i afla ID-ul și, opțional, în cazul în care stratul este de tip vectorial, pentru a calcula numărul total de entități spaiale. Pentru interacțiunea cu mediul QGIS, există o variabilă `:date:'iface'`, care reprezintă o instanță a clasei `QgsInterface`. Această interfață permite accesul la canvasul hărții, la meniuri, la barele de instrumente și la alte părți ale aplicației QGIS.

Pentru confortul utilizatorului, următoarele instrucțiuni sunt executate atunci când consola este pornită (în viitor, va fi posibil să stabilii comenzi inițiale suplimentare)

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

Pentru cei care folosesc des consola, ar putea fi utilă stabilirea unei comenzi rapide pentru deschiderea consolei (prin intermediul meniului *Setări* → *Configurare Comenzi Rapide...*)

1.3 Plugin-uri Python

QGIS permite îmbunătățirea funcționalităților sale, prin intermediul plugin-urilor. Acest lucru a fost inițial posibil numai cu ajutorul limbajului C. O dată cu adăugarea în QGIS a suportului pentru Python, a devenit posibilă folosirea de plugin-uri scrise în Python. Principalul avantaj față de plugin-urile în C constă în simplitatea distribuției (nu este necesară compilarea pentru fiecare platformă), iar dezvoltarea este mai ușoară.

De la momentul introducerii suportului pentru Python, au fost scrise multe plugin-uri, care acoperă diverse funcționalități. Instalatorul de plugin-uri facilitează utilizatorilor instalarea, actualizarea și eliminarea plugin-urilor Python. Parcurgeți pagina [Depozitele de Plugin-uri Python](#) pentru a descoperi diverse surse de plugin-uri.

Crearea de plugin-uri în Python este simplă, instrucțiuni detaliate găsindu-se în [:ref: developing_plugins](#).

1.4 Aplicații Python

Adesea, atunci când are loc procesarea unor date GIS, este recomandabilă crearea unor script-uri pentru automatizarea procesului, în locul repetării anevoioase a acelorai pași. Folosind PyQGIS, acest lucru este perfect posibil — importai modulul `qgis.core`, îl inițializezi, apoi sunteți gata de procesare.

Sau poate că doriți să creați o aplicație interactivă care utilizează unele funcționalități GIS — cum ar fi măsurarea anumitor date, exportarea unei hărți în format PDF sau orice altceva. Modulul `qgis.gui` aduce diverse componente GUI suplimentare, în special controlul grafic pentru canvas, care poate fi foarte ușor încorporat în aplicații, oferind suport pentru instrumentele de transfocare, deplasare și/sau pentru oricare alt instrument de hartă, personalizabil.

1.4.1 Using PyQGIS in custom application

Notă: *nu* utilizezi `qgis.py` ca nume pentru script-ul de test — în acest caz Python nu va fi capabil să importe legăturile.

First of all you have to import qgis module, set QGIS path where to search for resources — database of projections, providers etc. When you set prefix path with second argument set as `True`, QGIS will initialize all paths with standard dir under the prefix directory. Calling `initQgis()` function is important to let QGIS search for the available providers.

```
from qgis.core import *

# supply path to where is your qgis installed
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# load providers
QgsApplication.initQgis()
```

Acum puteți lucra cu API-ul QGIS — să încărcați straturile, să faceți unele prelucrări sau să startați un GUI cu un canvas pentru hartă. Posibilitățile sunt nelimitate :-)

When you are done with using QGIS library, call `exitQgis()` to make sure that everything is cleaned up (e.g. clear map layer registry and delete layers):

```
QgsApplication.exitQgis()
```

1.4.2 Rularea Aplicațiilor Personalizate

Trebuie să indicați sistemului dvs. unde să caute bibliotecile QGIS și modulele Python corespunzătoare, atunci când acestea nu se află într-o locație standard — altfel, Python vă va notifica:

```
>>> import qgis.core
ImportError: No module named qgis.core
```

Acest lucru se poate remedia prin setarea variabilei de mediu `PYTHONPATH`. În următoarele comenzi, `qgispath` ar trebui să fie înlocuit de calea instalării actuale de QGIS:

- în Linux: **`export PYTHONPATH=/qgispath/share/qgis/python`**
- în Windows: **`set PYTHONPATH=c:\qgispath\python`**

Dei calea către modulele PyQGIS este de acum cunoscută, ele depind totuși de bibliotecile `qgis_core` și `qgis_gui` (modulele Python servesc numai pentru intermedierea apelării). Calea către aceste biblioteci este de obicei necunoscută sistemului de operare, astfel că vei obține iarăși o eroare de import (mesajul putând varia în funcție de sistem):

```
>>> import qgis.core
ImportError: libqgis_core.so.1.5.0: cannot open shared object file: No such file or directory
```

Remediai acest lucru prin adăugarea directoarelor în care rezidă bibliotecile QGIS la calea de căutare a editorului de legături:

- în Linux: **export LD_LIBRARY_PATH=/qgispath/lib**
- în Windows: **set PATH=C:\qgispath;%PATH%**

Aceste comenzi pot fi puse într-un script bootstrap, care se va ocupa de pornire. Atunci când livrai aplicații personalizate folosind PyQGIS, există, de obicei, două variante:

- să cerei utilizatorului să instaleze QGIS pe platforma sa înainte de a instala aplicația dumneavoastră. Programul de instalare al aplicației ar trebui să caute locațiile implicite ale bibliotecilor QGIS și să permită utilizatorului setarea căii, în cazul în care ea nu poate fi găsită. Deși această abordare are avantajul de a fi mai simplă, este nevoie ca utilizatorul să parcurgă mai multe etape.
- să împachetați QGIS împreună cu aplicația dumneavoastră. Livrarea aplicației poate fi mai dificilă deoarece pachetul va fi foarte mare, dar utilizatorul va fi salvat de povara de a descărca și instala software suplimentar.

Cele două modele pot fi combinate - puteți distribui aplicația independentă pe Windows și Mac OS X, lăsând la îndemâna utilizatorului și a managerului de pachete instalarea QGIS pe Linux.

Încărcarea proiectelor

Uneori trebuie să încarci un proiect existent dintr-un plugin, sau (mai des) atunci când dezvoltai o aplicaie QGIS stand-alone în Python (see: *Aplicaii Python*).

To load a project into the current QGIS application you need a `QgsProject` instance() object and call its `read()` method passing to it a `QFileInfo` object that contains the path from where the project will be loaded:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt4 classes you will use in this script as shown below:
from qgis.core import QgsProject
from PyQt4.QtCore import QFileInfo
# Get the project instance
project = QgsProject.instance()
# Print the current project file name (might be empty in case no projects have been loaded)
print project.fileName
u'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
print project.fileName
u'/home/user/projects/my_other_qgis_project.qgs'
```

În cazul în care trebuie să faci unele modificări la proiect (de exemplu, să adăugai sau să elimini unele straturi), apoi să salvi modificările, poți apela metoda `write()` asupra instanței proiectului. De asemenea, metoda `write()` acceptă `QFileInfo`, opțional, care vă permite să specificați o cale în care va fi salvat proiectul:

```
# Save the project to the same
project.write()
# ... or to a new file
project.write(QFileInfo('/home/user/projects/my_new_qgis_project.qgs'))
```

Ambele funcții, `read()` și `write()`, returnează o valoare booleană, pe care o puteți folosi pentru a verifica dacă operaia a avut succes.

Încărcarea Straturilor

Haidei să deschidem mai multe straturi cu date. QGIS recunoaște straturile vectoriale și pe cele de tip raster. În plus, sunt disponibile și tipurile de straturi personalizate, dar pe acestea nu le vom discuta aici.

3.1 Straturile Vectoriale

To load a vector layer, specify layer's data source identifier, name for the layer and provider's name:

```
layer = QgsVectorLayer(data_source, layer_name, provider_name)
if not layer.isValid():
    print "Layer failed to load!"
```

Identificatorul sursei de date reprezintă un ir specific pentru fiecare furnizor de date vectoriale în parte. Numele stratului se va afla în lista straturilor. Este important să se verifice dacă stratul a fost încărcat cu succes. În cazul neîncărcării cu succes, va fi returnată o instanță de strat nevalid.

Cea mai rapidă cale de a deschide și de a afla un strat vectorial în QGIS are loc prin utilizarea funcției `addVectorLayer` din clasa `QgisInterface`:

```
layer = iface.addVectorLayer("/path/to/shapefile/file.shp", "layer_name_you_like", "ogr")
if not layer:
    print "Layer failed to load!"
```

Astfel se creează un nou strat care va fi adăugat într-un singur pas în registrul de straturi al hărții (ceea ce-l va face să apară în lista straturilor). Funcția returnează instanța stratului, sau *None* dacă stratul nu a putut fi încărcat.

Lista de mai jos arată modul de accesare a diverselor surse de date, cu ajutorul furnizorilor de date vectoriale:

- OGR library (shapefiles and many other file formats) — data source is the path to the file

```
vlayer = QgsVectorLayer("/path/to/shapefile/file.shp", "layer_name_you_like", "ogr")
```

- PostGIS database — data source is a string with all information needed to create a connection to PostgreSQL database. `QgsDataSourceURI` class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn't available.

```
uri = QgsDataSourceURI()
# set host name, port, database name, username and password
uri.setConnection("localhost", "5432", "dbname", "johnny", "xxx")
# set database schema, table name, geometry column and optionally
# subset (WHERE clause)
uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

vlayer = QgsVectorLayer(uri.uri(), "layer_name_you_like", "postgres")
```

- CSV or other delimited text files — to open a file with a semicolon as a delimiter, with field “x” for x-coordinate and field “y” with y-coordinate you would use something like this

```
uri = "/some/path/file.csv?delimiter=%s&xField=%s&yField=%s" % (";", "x", "y")
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "delimitedtext")
```

Note: from QGIS version 1.7 the provider string is structured as a URL, so the path must be prefixed with *file://*. Also it allows WKT (well known text) formatted geometries as an alternative to “x” and “y” fields, and allows the coordinate reference system to be specified. For example

```
uri = "file:///some/path/file.csv?delimiter=%s&crs=epsg:4723&wktField=%s" % (";", "shape")
```

- GPX files — the “gpx” data provider reads tracks, routes and waypoints from gpx files. To open a file, the type (track/route/waypoint) needs to be specified as part of the url

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "gpx")
```

- Spatialite database — supported from QGIS v1.1. Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier

```
uri = QgsDataSourceURI()
uri.setDatabase('/home/martin/test-2.3.sqlite')
schema = ''
table = 'Towns'
geom_column = 'Geometry'
uri.setDataSource(schema, table, geom_column)

display_name = 'Towns'
vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
```

- MySQL WKB-based geometries, through OGR — data source is the connection string to the table

```
uri = "MySQL:dbname,host=localhost,port=3306,user=root,password=xxx|layername=my_table"
vlayer = QgsVectorLayer(uri, "my_table", "ogr")
```

- WFS connection: the connection is defined with a URI and using the WFS provider

```
uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=union&version=1.0.0&request=GetFeature"
vlayer = QgsVectorLayer("my_wfs_layer", "WFS")
```

The uri can be created using the standard `urllib` library.

```
params = {
    'service': 'WFS',
    'version': '1.0.0',
    'request': 'GetFeature',
    'typename': 'union',
    'srsname': "EPSG:23030"
}
uri = 'http://localhost:8080/geoserver/wfs?' + urllib.unquote(urllib.urlencode(params))
```

3.2 Straturile Raster

For accessing raster files, GDAL library is used. It supports a wide range of file formats. In case you have troubles with opening some files, check whether your GDAL has support for the particular format (not all formats are available by default). To load a raster from a file, specify its file name and base name

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
if not rlayer.isValid():
    print "Layer failed to load!"
```

Similar straturilor vectoriale, straturile raster pot fi încărcate cu ajutorul funcției `addRasterLayer` a clasei `QgisInterface`:

```
iface.addRasterLayer("/path/to/raster/file.tif", "layer_name_you_like")
```

Astfel se creează un nou strat care se adaugă la registrul de straturi al hărții într-un singur pas (făcându-l să apară în lista straturilor).

Raster layers can also be created from a WCS service.

```
layer_name = 'modis'
uri = QgsDataSourceURI()
uri.setParam('url', 'http://demo.mapserver.org/cgi-bin/wcs')
uri.setParam("identifier", layer_name)
rlayer = QgsRasterLayer(str(uri.encodedUri()), 'my_wcs_layer', 'wcs')
```

setările URI detaliate pot fi găsite în [documentația furnizorului](#)

Alternatively you can load a raster layer from WMS server. However currently it's not possible to access `GetCapabilities` response from API — you have to know what layers you want

```
urlWithParams = 'url=http://wms.jpl.nasa.gov/wms.cgi&layers=global_mosaic&styles=pseudo&format=im
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
    print "Layer failed to load!"
```

3.3 Registrul Straturilor de Hartă

Dacă dorești să utilizezi straturile deschise pentru randare, nu uitați să le adăugați în registrul straturilor de hartă. Acest registru înregistrează proprietatea asupra straturilor, acestea putând fi accesate ulterior din oricare parte a aplicației după ID-ul lor unic. Atunci când un strat este eliminat din registru, va fi i ters totodată.

Adding a layer to the registry

```
QgsMapLayerRegistry.instance().addMapLayer(layer)
```

Layers are destroyed automatically on exit, however if you want to delete the layer explicitly, use

```
QgsMapLayerRegistry.instance().removeMapLayer(layer_id)
```

For a list of loaded layers and layer ids, use

```
QgsMapLayerRegistry.instance().mapLayers()
```

TODO: More about map layer registry?

Utilizarea straturilor raster

Această secțiune enumeră diverse operațiuni pe care le puteți efectua cu straturile raster.

4.1 Detaliile stratului

Un strat raster constă într-una sau mai multe benzi raster - cu referire la rastere cu o singură bandă sau multibandă. O bandă reprezintă o matrice de valori. Imaginea color obișnuită (cum ar fi o fotografie aeriană) este un format raster cu o bandă roie, una albastră și una verde. Straturile cu bandă unică reprezintă, de obicei, fie variabile continue (de exemplu, elevația) fie variabile discrete (cum ar fi utilizarea terenului). În unele cazuri, un strat raster vine cu o paletă, iar valorile rasterului fac referire la culorile stocate în paletă:

```
rlayer.width(), rlayer.height()
(812, 301)
rlayer.extent()
<qgis._core.QgsRectangle object at 0x000000000F8A2048>
rlayer.extent().toString()
u'12.095833,48.552777 : 18.863888,51.056944'
rlayer.rasterType()
2 # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
rlayer.bandCount()
3
rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
rlayer.hasPyramids()
False
```

4.2 Drawing Style

When a raster layer is loaded, it gets a default drawing style based on its type. It can be altered either in raster layer properties or programmatically. The following drawing styles exist:

In-dex	Constant: QgsRasterLater.X	Comment
1	SingleBandGray	Single band image drawn as a range of gray colors
2	SingleBandPseudoColor	Single band image drawn using a pseudocolor algorithm
3	PalettedColor	“Palette” image drawn using color table
4	PalettedSingleBandGray	“Palette” layer drawn in gray scale
5	PalettedSingleBandPseudo-Color	“Palette” layer drawn using a pseudocolor algorithm
7	MultiBandSingleBandGray	Layer containing 2 or more bands, but a single band drawn as a range of gray colors
8	MultiBandSingle-BandPseudoColor	Layer containing 2 or more bands, but a single band drawn using a pseudocolor algorithm
9	MultiBandColor	Layer containing 2 or more bands, mapped to RGB color space.

To query the current drawing style:

```
rlayer.renderer().type()
u'singlebandpseudocolor'
```

Straturile cu o singură bandă raster pot fi desenate fie în nuane de gri (valori mici = negru, valori ridicate = alb), sau cu un algoritm cu pseudocolori, care atribuie culori valorilor din banda singulară. Rasterele cu o singură bandă pot fi desenate folosindu-se propria paletă. Straturile multibandă sunt, de obicei, desenate prin maparea benzilor la culori RGB. Altă posibilitate este de a utiliza doar o singură bandă pentru desenarea în tonuri de gri sau cu pseudocolori.

Următoarele seciuni explică modul în care se poate interoga și modifica stilul de desenare al stratului. După efectuarea schimbărilor, ai putea forța actualizarea suprafeei hărții, a se vedea [Recitirea straturilor](#).

DE EFECTUAT: îmbunătățiri de contrast, de transparență (date nule), min/max definit de utilizator, statistici bandă

4.2.1 Rastere cu o singură bandă

They are rendered in gray colors by default. To change the drawing style to pseudocolor:

```
# Check the renderer
rlayer.renderer().type()
u'singlebandgray'
rlayer.setDrawingStyle("SingleBandPseudoColor")
# The renderer is now changed
rlayer.renderer().type()
u'singlebandpseudocolor'
# Set a color ramp hader function
shader_func = QgsColorRampShader()
rlayer.renderer().shader().setRasterShaderFunction(shader_func)
```

The `PseudoColorShader` is a basic shader that highlights low values in blue and high values in red. There is also `ColorRampShader` which maps the colors as specified by its color map. It has three modes of interpolation of values:

- liniar (INTERPOLAT): culoarea rezultată fiind interpolată liniar, de la intrările hărții de culori, în sus sau în jos față de valoarea înscrisă în harta de culori
- discret (DISCRET): culorile folosite fiind cele cu o valoare egală sau mai mare față de cele din harta de culori
- exact (EXACT): culoarea nu este interpolată, desenându-se doar pixelii cu o valoare egală cu cea introdusă în harta de culori

To set an interpolated color ramp shader ranging from green to yellow color (for pixel values from 0 to 255):

```
rlayer.renderer().shader().setRasterShaderFunction(QgsColorRampShader())
lst = [QgsColorRampShader.ColorRampItem(0, QColor(0, 255, 0)), \
```



```

    QgsColorRampShader.ColorRampItem(255, QColor(255, 255, 0))]
fcn = rlayer.renderer().shader().rasterShaderFunction()
fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
fcn.setColorRampItemList(lst)

```

To return back to default gray levels, use:

```

rlayer.setDrawingStyle('SingleBandGray')

```

4.2.2 Rastere multibandă

În mod implicit, QGIS mapează primele trei benzi la valori de rou, verde i albastru, pentru a crea o imagine color (desenată în stilul `MultiBandColor`. În unele cazuri, ai putea dori să suprascrii aceste setări. Următorul cod inversează banda roie (1) cu cea verde (2):

```

rlayer.setDrawingStyle('MultiBandColor')
rlayer.renderer().setGreenBand(1)
rlayer.setRedBand(2)

```

4.3 Recitirea straturilor

Dacă schimbi simbologia stratului i ai vrea să vă asigurai că schimbările sunt imediat vizibile pentru utilizator, putei apela aceste metode

```

if hasattr(layer, "setCacheImage"):
    layer.setCacheImage(None)
layer.triggerRepaint()

```

Primul apel garantează că imaginea din cache a stratului este tearsă în cazul în care cache-ul este activat. Această funcionalitate este disponibilă începând de la QGIS 1.4, în versiunile anterioare această funcie neexistând — pentru a fi siguri de cod, că funcionează cu toate versiunile de QGIS, vom verifica în primul rând dacă metoda există.

Al doilea apel emite semnalul care va foră orice suport de hartă, care conine stratul, să emită o reîmprospătare.

Aceste comenzi nu funcionează în cazul straturilor raster WMS. În acest caz, trebuie să specifici în mod explicit

```

layer.dataProvider().reloadData()
layer.triggerRepaint()

```

În cazul în care s-a schimbat simbologia stratului (a se vedea seciunea despre straturile raster i cele vectoriale cu privire la modul cum se face acest lucru), ai putea foră QGIS să actualizeze simbologia din lista straturilor (legendă). Acest lucru poate fi realizat după cum urmează (iface este o instană a `QgisInterface`)

```

iface.legendInterface().refreshLayerSymbology(layer)

```

4.4 Interogarea valorilor

Pentru a face, la un moment dat, o interogare asupra valorilor din benzile stratului raster

```

ident = rlayer.dataProvider().identify(QgsPoint(15.30, 40.98), \
    QgsRaster.IdentifyFormatValue)
if ident.isValid():
    print ident.results()

```

În acest caz, metoda `results` returnează un dicționar, cu indicii benzii ca i chei, i valorile benzii ca valori.

```
{1: 17, 2: 220}
```

Utilizarea straturilor vectoriale

Această seciune rezumă diferitele acțiuni care pot fi efectuate asupra straturilor vectoriale.

5.1 Retrieving informations about attributes

You can retrieve informations about the fields associated with a vector layer by calling `pendingFields()` on a `QgsVectorLayer` instance:

```
# "layer" is a QgsVectorLayer instance
for field in layer.pendingFields():
    print field.name(), field.typeName()
```

5.2 Selectarea entităților

In QGIS desktop, features can be selected in different ways, the user can click on a feature, draw a rectangle on the map canvas or use an expression filter. Selected features are normally highlighted in a different color (default is yellow) to draw user's attention on the selection. Sometimes can be useful to programmatically select features or to change the default color.

Pentru a schimba culoarea de selecție puteți utiliza metoda `setSelectionColor()` din `QgsMapCanvas`, așa cum se arată în exemplul următor:

```
iface.mapCanvas().setSelectionColor( QColor("red") )
```

Pentru a adăuga entitățile în lista de entități selectate ale unui strat dat, puteți apela `setSelectedFeatures()`, pasându-i lista de ID-uri a entităților:

```
# Get the active layer (must be a vector layer)
layer = iface.activeLayer()
# Get the first feature from the layer
feature = layer.getFeatures().next()
# Add this features to the selected list
layer.setSelectedFeatures([feature.id()])
```

Pentru a anula selecția, transmiteți doar o listă vidă:

```
layer.setSelectedFeatures([])
```

5.3 Iterații în straturile vectoriale

Parcurgerea elementelor dintr-un strat vectorial este una dintre cele mai obișnuite activități. Mai jos este prezentat un exemplu de cod de bază, simplu, pentru a efectua această sarcină și care arată unele informații despre fiecare

entitate spațială. Variabila `layer` se consideră a conține un obiect `QgsVectorLayer`

```
iter = layer.getFeatures()
for feature in iter:
    # retrieve every feature with its geometry and attributes
    # fetch geometry
    geom = feature.geometry()
    print "Feature ID %d: " % feature.id()

    # show some information about the feature
    if geom.type() == Qgs.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == Qgs.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == Qgs.Polygon:
        x = geom.asPolygon()
        numPts = 0
        for ring in x:
            numPts += len(ring)
        print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
        print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

    # attrs is a list. It contains all the attribute values of this feature
    print attrs
```

5.3.1 Accesarea atributelor

Atributele pot fi apelate după numele lor.

```
print feature['name']
```

Alternativ, atributele pot fi menționate de index. Acesta va fi un pic mai rapid decât prin folosirea numelui. De exemplu, pentru a obține primul atribut:

```
print feature[0]
```

5.3.2 Parcurgerea entităților selectate

dacă aveți nevoie doar de entitățile selectate, puteți utiliza metoda `selectedFeatures()` din stratul vectorial:

```
selection = layer.selectedFeatures()
print len(selection)
for feature in selection:
    # do whatever you need with the feature
```

O altă opțiune o constituie metoda `Processing features()`:

```
import processing
features = processing.features(layer)
for feature in features:
    # do whatever you need with the feature
```

În mod implicit se vor parcurge toate entitățile stratului, în cazul în care nu există o selecție, sau, în caz contrar, doar entitățile selectate. Rețineți că acest comportament poate fi schimbat în opțiunile `Processing`, pentru a ignora selecțiile.

5.3.3 Parcurgerea unui subset de entităi

Dacă dorești să parcurgi un anumit subset de entități dintr-un strat, cum ar fi cele dintr-o anumită zonă, trebuie să adăugai un obiect `QgsFeatureRequest` la apelul funcției `getFeatures()`. Iată un exemplu

```
request = QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for feature in layer.getFeatures(request):
    # do whatever you need with the feature
```

Dacă ai nevoie de un filtru pe bază de atribut în locul unui spațial (sau în plus față de acesta), aa cum se vede în exemplul de mai sus, poți construi un obiect `QgsExpression` și să-i transmiți constructorului `QgsFeatureRequest`. Iată un exemplu

```
# The expression will filter the features where the field "location_name" contains
# the word "Lake" (case insensitive)
exp = QgsExpression('location_name ILIKE \'%Lake%\'')
request = QgsFeatureRequest(exp)
```

Cererea poate fi utilizată pentru a defini datele cerute pentru fiecare entitate, astfel încât iteratorul să înapoieze toate entitățile, dar să returneze datele parțiale pentru fiecare dintre ele.

```
# Only return selected fields
request.setSubsetOfAttributes([0,2])
# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.pendingFields())
# Don't return geometry objects
request.setFlags(QgsFeatureRequest.NoGeometry)
```

Tip: If you only need a subset of the attributes or you don't need the geometry informations, you can significantly increase the **speed** of the features request by using `QgsFeatureRequest.NoGeometry` flag or specifying a subset of attributes (possibly empty) like shown in the example above.

5.4 Modificarea straturilor vectoriale

Cei mai mulți dintre furnizorii de date vectoriale suportă editarea datelor stratului. Uneori, aceștia acceptă doar un subset restrâns de acțiuni de editare. Utilizai funcția `capabilities()` pentru a afla care set de funcții este disponibil

```
caps = layer.dataProvider().capabilities()
```

Utilizând oricare dintre următoarele metode de editare a straturilor vectoriale, schimbările sunt efectuate direct în depozitul de date (un fișier, o bază de date etc). În cazul în care dorești să faci doar schimbări temporare, treceți la secțiunea următoare, care explică efectuarea *modificărilor cu ajutorul tamponului de editare*.

Note: Dacă lucrezi în interiorul QGIS (fie din consola fie printr-un plugin), ar putea fi necesar să forai o redesenare a canvasului hărții, pentru a vedea modificările aduse geometriei, stilului sau atributelor:

```
# If caching is enabled, a simple canvas refresh might not be sufficient
# to trigger a redraw and you must clear the cached image for the layer
if iface.mapCanvas().isCachingEnabled():
    layer.setCacheImage(None)
else:
    iface.mapCanvas().refresh()
```

5.4.1 Adăugarea entităților

Create some `QgsFeature` instances and pass a list of them to provider's `addFeatures()` method. It will return two values: result (true/false) and list of added features (their ID is set by the data store)

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature()
    feat.addAttribute(0, 'hello')
    feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
    (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

5.4.2 tergerea entităților

Pentru a terge unele entități, e suficientă furnizarea unei liste cu ID-uri

```
if caps & QgsVectorDataProvider.DeleteFeatures:
    res = layer.dataProvider().deleteFeatures([5, 10])
```

5.4.3 Modificarea entităților

Este posibilă, fie schimbarea geometriei unei entități, fie schimbarea unor atribute. În următorul exemplu are loc mai întâi schimbarea valorilor atributelor cu indexul 0 sau 1, iar mai apoi se schimbă geometria entității

```
fid = 100 # ID of the feature we will modify

if caps & QgsVectorDataProvider.ChangeAttributeValues:
    attrs = { 0 : "hello", 1 : 123 }
    layer.dataProvider().changeAttributeValues({ fid : attrs })

if caps & QgsVectorDataProvider.ChangeGeometries:
    geom = QgsGeometry.fromPoint(QgsPoint(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

Tip: Dacă trebuie doar să schimbi geometriile, ai putea lua în considerare utilizarea `QgsVectorLayerEditUtils` care oferă unele dintre metodele utile pentru a edita geometrii (traducere, introducere sau mutare vertex etc.)

5.4.4 Adăugarea și eliminarea câmpurilor

Pentru a adăuga câmpuri (atribute), trebuie să specifici o listă de definiții pentru acestea. Pentru tergerea de câmpuri e suficientă furnizarea unei liste de indici pentru câmpuri.

```
if caps & QgsVectorDataProvider.AddAttributes:
    res = layer.dataProvider().addAttributes([QgsField("mytext", QVariant.String), QgsField("myint", QVariant.Int)])

if caps & QgsVectorDataProvider.DeleteAttributes:
    res = layer.dataProvider().deleteAttributes([0])
```

După adăugarea sau eliminarea câmpurilor din furnizorul de date, câmpurile stratului trebuie să fie actualizate, deoarece modificările nu se propagă automat.

```
layer.updateFields()
```

5.5 Modificarea straturi vectoriale prin editarea unui tampon de memorie

Când editai vectori în aplicaia QGIS, în primul rând, trebuie să comutai în modul de editare pentru stratul în care lucrai, apoi să efectuai modificări pe care, în cele din urmă, să le salvi (sau să le anulai). Modificările nu vor fi scrise până când nu sunt salvate — ele rezidând în memorie, în tamponul de editare al stratului. De asemenea, este posibilă utilizarea programatică a acestei funcționalități — aceasta fiind doar o altă metodă pentru editarea straturilor vectoriale, care completează utilizarea directă a furnizorilor de date. Utilizai această opțiune atunci când furnizai unele instrumente GUI pentru editarea straturilor vectoriale, permiând utilizatorului să decidă dacă să salveze/anuleze, i punându-i la dispoziție facilitățile de undo/redo. Atunci când salvi modificările, acestea vor fi transferate din memoria tampon de editare în furnizorul de date.

To find out whether a layer is in editing mode, use `isEditing()` — the editing functions work only when the editing mode is turned on. Usage of editing functions

```
# add two features (QgsFeature instances)
layer.addFeatures([feat1, feat2])
# delete a feature with specified ID
layer.deleteFeature(fid)

# set new geometry (QgsGeometry instance) for a feature
layer.changeGeometry(fid, geometry)
# update an attribute with given field index (int) to given value (QVariant)
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QVariant.String))
# remove a field
layer.deleteAttribute(fieldIndex)
```

Pentru ca undo/redo să funcționeze în mod corespunzător, apelurile de mai sus trebuie să fie înglobate în comenzi undo. (Dacă nu vă pasă de undo/redo i doriți să stocați imediat modificările, atunci vei avea o sarcină mai ușoară prin :ref: folosirea <editorului> furnizorului de date.) Cum să utilizai funcționalitatea undo

```
layer.beginEditCommand("Feature triangulation")

# ... call layer's editing methods ...

if problem_occurred:
    layer.destroyEditCommand()
    return

# ... more editing ...

layer.endEditCommand()
```

`beginEditCommand()` va crea o comandă internă “activă” i va înregistra modificările ulterioare din stratul vectorial. Cu apelul către `endEditCommand()` comanda este împinsă pe stiva undo, iar utilizatorul va putea efectua undo/redo prin GUI. În cazul în care ceva nu a mers bine pe timpul efectuării schimbărilor, metoda `destroyEditCommand()` va elimina comanda i va da înapoi toate modificările făcute pe perioada când această comandă a fost activă.

To start editing mode, there is `startEditing()` method, to stop editing there are `commitChanges()` and `rollback()` — however normally you should not need these methods and leave this functionality to be triggered by the user.

5.6 Crearea unui index spaial

Indecii spaiali pot îmbunătăți dramatic performanța codului dvs, în cazul în care este nevoie să interogai frecvent un strat vectorial. Imaginați-vă, de exemplu, că scriei un algoritm de interpolare, i că, pentru o anumită locație, trebuie

să afli cele mai apropiate 10 puncte dintr-un strat, în scopul utilizării acelor puncte în calculul valorii interpolate. Fără un index spaial, singura modalitate pentru QGIS de a găsi cele 10 puncte, este de a calcula distanța tuturor punctelor față de locaia specificată și apoi de a compara aceste distanțe. Această sarcină poate fi mare consumatoare de timp, mai ales în cazul în care trebuie să fie repetată pentru mai multe locații. Dacă pentru stratul respectiv există un index spaial, operațiunea va fi mult mai eficientă.

Gândiți-vă la un strat fără index spaial ca la o carte de telefon în care numerele de telefon nu sunt ordonate sau indexate. Singura modalitate de a afla numărul de telefon al unei anumite persoane este de a citi toate numerele, începând cu primul, până când îl găsiți.

Spatial indexes are not created by default for a QGIS vector layer, but you can create them easily. This is what you have to do.

1. creare index spaial — următorul cod creează un index vid

```
index = QgsSpatialIndex()
```

2. add features to index — index takes `QgsFeature` object and adds it to the internal data structure. You can create the object manually or use one from previous call to provider's `nextFeature()`

```
index.insertFeature(featur)
```

3. o dată ce ai introdus valori în indexul spaial, poți efectua unele interogări

```
# returns array of feature IDs of five nearest features
nearest = index.nearestNeighbor(QgsPoint(25.4, 12.7), 5)
```

```
# returns array of IDs of features which intersect the rectangle
intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
```

5.7 Scrierea straturilor vectoriale

Puteti scrie în fierele conținând straturi vectoriale folosind clasa `QgsVectorFileWriter`. Aceasta acceptă orice alt tip de fiier vector care suportă OGR (fiere shape, GeoJSON, KML și altele).

Există două posibilități de a exporta un strat vectorial:

- dintr-o instanță a `QgsVectorLayer`

```
error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_shapes.shp", "CP1250", None, "ESRI")
```

```
if error == QgsVectorFileWriter.NoError:
    print "success!"
```

```
error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSON")
```

```
if error == QgsVectorFileWriter.NoError:
    print "success again!"
```

The third parameter specifies output text encoding. Only some drivers need this for correct operation - shapefiles are one of those — however in case you are not using international characters you do not have to care much about the encoding. The fourth parameter that we left as `None` may specify destination CRS — if a valid instance of `QgsCoordinateReferenceSystem` is passed, the layer is transformed to that CRS.

For valid driver names please consult the [supported formats by OGR](#) — you should pass the value in the “Code” column as the driver name. Optionally you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes — look into the documentation for full syntax.

- direct din entitățile


```

# define fields for feature attributes. A list of QgsField objects is needed
fields = [QgsField("first", QVariant.Int),
          QgsField("second", QVariant.String)]

# create an instance of vector file writer, which will create the vector file.
# Arguments:
# 1. path to new file (will fail if exists already)
# 2. encoding of the attributes
# 3. field map
# 4. geometry type - from WKBTYPPE enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI SH

if writer.hasError() != QgsVectorFileWriter.NoError:
    print "Error when creating shapefile: ", writer.hasError()

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk (optional)
del writer

```

5.8 Furnizorul de memorie

Furnizorul de memorie este destinat, în principal, dezvoltatorilor de plugin-uri sau de aplicații terțe. El nu stochează date pe disc, permiind dezvoltatorilor să-l folosească ca pe un depozit rapid pentru straturi temporare.

Furnizorul suportă câmpuri de tip string, int sau double.

Furnizorul de memorie suportă, de asemenea, indexarea spațială, care este activată prin apelarea furnizorului funcției `createSpatialIndex()`. O dată ce indexul spațial este creat, vei fi capabil de a parcurge mai rapid entitățile, în interiorul unor regiuni mai mici (din moment ce nu este necesar să traversezi toate entitățile, ci doar pe cele din dreptunghiul specificat).

Un furnizor de memorie este creat prin transmiterea "memoriei" ca ir furnizor către constructorul `QgsVectorLayer`.

Constructorul are, de asemenea, un URI care definește unul din următoarele tipuri de geometrie a stratului: "Point", "LineString", "Polygon", "MultiPoint", "MultiLineString" sau "MultiPolygon".

URI poate specifica, de asemenea, sistemul de coordonate de referință, câmpurile, precum și indexarea furnizorului de memorie. Sintaxa este:

crs=definiie Specificai sistemul de referință de coordonate, unde definiția poate fi oricare din formele acceptate de: `QgsCoordinateReferenceSystem.createFromString()`

index=yes Specificai dacă furnizorul va utiliza un index spațial.

field=nume:tip(lungime,precizie) Specificai un atribut al stratului. Atributul are un nume și, opțional, un tip (integer, double sau string), lungime și precizie. Pot exista mai multe definiții de câmp.

Următorul exemplu de URI încorporează toate aceste opțiuni

```
"Point?crs=epsg:4326&field=id:integer&field=name:string(20)&index=yes"
```

Următorul exemplu de cod ilustrează crearea și popularea unui furnizor de memorie

```

# create layer
vl = QgsVectorLayer("Point", "temporary_points", "memory")
pr = vl.dataProvider()

# add fields
pr.addAttributes([QgsField("name", QVariant.String),
                  QgsField("age", QVariant.Int),
                  QgsField("size", QVariant.Double)])
vl.updateFields() # tell the vector layer to fetch changes from the provider

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])

# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()

```

În cele din urmă, să verificăm dacă totul a mers bine

```

# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum()

# iterate over features
f = QgsFeature()
features = vl.getFeatures()
for f in features:
    print "F:", f.id(), f.attributes(), f.geometry().asPoint()

```

5.9 Aspectul (simbologia) straturilor vectoriale

Când un strat vector este randat, aspectul datelor este dat de **render** i de **simbolurile** asociate stratului. Simbolurile sunt clase care au grija de reprezentarea vizuală a tuturor entităților, în timp ce un render determină ce simbol va fi folosit doar pentru anumite entități.

Tipul de render pentru un strat oarecare poate fi obținut astfel:

```
renderer = layer.rendererV2()
```

i cu acea referință, să explorăm un pic

```
print "Type:", rendererV2.type()
```

Există mai multe tipuri de rendere disponibile în biblioteca de bază a QGIS:

Tipul	Clasa	Descrierea
singleSymbol	QgsSingleSymbolRendererV2	Asociază tuturor entităților același simbol
categorizedSymbol	QgsCategorizedSymbolRendererV2	Asociază entităților un simbol diferit, în funcție de categorie
graduatedSymbol	QgsGraduatedSymbolRendererV2	Asociază fiecărei entități un simbol diferit pentru fiecare gamă de valori

Ar mai putea exista, de asemenea, unele tipuri de randare personalizate, aa că niciodată să nu presupunem că există doar aceste tipuri. Puteți interoga singleton-ul `QgsRendererV2Registry` pentru a afla tipurile de rendere disponibile în prezent:

```
QgsRendererV2Registry.instance().renderersList()
# Prints:
[u' singleSymbol',
 u' categorizedSymbol',
 u' graduatedSymbol',
 u' RuleRenderer',
 u' pointDisplacement',
 u' invertedPolygonRenderer',
 u' heatmapRenderer']
```

Este posibilă obținerea conținutului renderului sub formă de text — lucru util pentru depanare

```
print rendererV2.dump()
```

5.9.1 Render cu Simbol Unic

Puteți obține simbolul folosit pentru randare apelând metoda `symbol()`, i-l puteți schimba cu ajutorul metodei `setSymbol()` (notă pentru dezvoltatorii C++: renderul devine proprietarul simbolului.)

Puteți schimba simbolul utilizat de un strat vectorial, particular, prin apelarea `setSymbol()` și transmiterea unei instanțe corespunzătoare de instanță simbol. Simbolurile pentru straturile de tip *punct*, *linie* și *poligon* pot fi create prin apelarea funcției `createSimple()` din clasele corespunzătoare, `QgsMarkerSymbolV2`, `QgsLineSymbolV2` și `QgsFillSymbolV2`.

Dicționarul transmis către `createSimple()` stabilește proprietățile de stil ale simbolului.

For example you can change the symbol used by a particular **point** layer by calling `setSymbol()` passing an instance of a `QgsMarkerSymbolV2` as in the following code example:

```
symbol = QgsMarkerSymbolV2.createSimple({'name': 'square', 'color': 'red'})
layer.rendererV2().setSymbol(symbol)
```

nume: indică forma markerului, aceasta putând fi oricare dintre următoarele:

- cerc
- pătrat
- dreptunghi
- diamant
- pentagon
- triunghi
- triunghi echilateral
- stea
- stea_regulată
- săgeată
- vârf_de_săgeată_plin

5.9.2 Render cu Simboluri Categorisite

Puteți interoga și seta numele atributului care este folosit pentru clasificare: folosiți metodele `classAttribute()` și `setClassAttribute()`.

Pentru a obține o listă de categorii

```
for cat in rendererV2.categories():
    print "%s: %s :: %s" % (cat.value().toString(), cat.label(), str(cat.symbol()))
```

În cazul în care `value()` reprezintă valoarea utilizată pentru discriminare între categorii, `label()` este un text utilizat pentru descrierea categoriei iar metoda `symbol()` returnează simbolul asignat.

Renderul, de obicei, stochează atât simbolul original cât și gamele de culoare care au fost utilizate pentru clasificare: metodele `sourceColorRamp()` și `sourceSymbol()`.

5.9.3 Render cu Simboluri Graduale

Acest render este foarte similar cu renderul cu simbol clasificat, descris mai sus, dar în loc de o singură valoare de atribut per clasă el lucrează cu intervale de valori, putând fi, astfel, utilizat doar cu atribute numerice.

Pentru a afla mai multe despre gamele utilizate în render

```
for ran in rendererV2.ranges():
    print "%f - %f: %s %s" % (
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
        str(ran.symbol())
    )
```

putei folosi din nou `classAttribute()` pentru a afla numele atributului de clasificare, metodele `sourceSymbol()` și `sourceColorRamp()`. În plus, există metoda `mode()` care determină modul în care au fost create gamele: folosind intervale egale, cuantile sau o altă metodă.

Dacă doriți să creați propriul render cu simbol gradual, puteți face acest lucru așa cum este ilustrat în fragmentul de mai jos (care creează un simplu aranjament cu două clase)

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
    myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRendererV2(myRenderer)
QgsMapLayerRegistry.instance().addMapLayer(myVectorLayer)
```

5.9.4 Lucrul cu Simboluri

Pentru reprezentarea simbolurilor există clasa de bază `QgsSymbolV2`, având trei clase derivate:

- `QgsMarkerSymbolV2` — pentru entități de tip punct
- `QgsLineSymbolV2` — pentru entități de tip linie
- `QgsFillSymbolV2` — pentru entități de tip poligon

Fiecare simbol este format din unul sau mai multe straturi (clase derivate din `QgsSymbolLayerV2`). Straturile simbolului realizează în mod curent randarea, clasa simbolului servind doar ca un container pentru acestea.

Având o instanță a unui simbol (de exemplu, de la un render), este posibil să o explorăm: metoda `type()` spunându-ne dacă acesta este un marker, o linie sau un simbol de umplere. Există și metoda `dump()` care returnează o scurtă descriere a simbolului. Pentru a obține o listă a straturilor simbolului

```
for i in xrange(symbol.symbolLayerCount()):
    lyr = symbol.symbolLayer(i)
    print "%d: %s" % (i, lyr.layerType())
```

Pentru a afla culoarea simbolului folosii metoda `color()`, iar pentru a schimba culoarea `setColor()`. În cazul simbolurilor marker, în plus, puteți interoga pentru dimensiunea simbolului și unghiul de rotație cu metodele `size()` și `angle()`, iar pentru simbolurile linie există metoda `width()` care returnează lăimea liniei.

Dimensiunea și lăimea sunt în milimetri, în mod implicit, iar unghiurile sunt în grade.

Lucrul cu Straturile Simbolului

Aa cum s-a arătat mai înainte, straturile simbolului (subclase ale `QgsSymbolLayerV2`), determină aspectul entităților. Există mai multe clase de strat simbol de bază, pentru uzul general. Este posibilă implementarea unor noi tipuri de strat simbol și, astfel, personalizarea în mod arbitrar a modului în care vor fi randate entitățile. Metoda `layerType()` identifică în mod unic clasa stratului simbol — tipurile de straturi simbol de bază și implicite sunt `SimpleMarker`, `SimpleLine` și `SimpleFill`.

Puteți obține, în modul următor, o listă completă a tipurilor de straturi pe care le puteți crea pentru o anumită clasă de simboluri

```
from qgis.core import QgsSymbolLayerV2Registry
myRegistry = QgsSymbolLayerV2Registry.instance()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbolV2.Marker):
    print item
```

Rezultat

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

clasa `QgsSymbolLayerV2Registry` gestionează o bază de date a tuturor tipurilor de straturi simbol disponibile.

Pentru a accesa datele stratului simbol, folosii metoda `properties()` care returnează un dicționar cu valorile ale proprietăților care îi determină aparența. Fiecare tip de strat simbol are un set specific de proprietăți pe care le utilizează. În plus, există metodele generice `color()`, `size()`, `angle()`, `width()` împreună cu cu omologii lor de setare. Desigur, mărimea și unghiul sunt disponibile doar pentru straturi simbol de tip marker iar lăimea pentru straturi simbol de tip linie.

Crearea unor Tipuri Personalizate de Straturi pentru Simboluri

Imaginați-vă că ai dori să personalizai modul în care se randează datele. Vă puteți crea propria dvs. clasă de strat de simbol, care va desena entitățile exact așa cum doriți. Iată un exemplu de marker care desenează cercuri roșii cu o rază specificată

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

    def __init__(self, radius=4.0):
        QgsMarkerSymbolLayerV2.__init__(self)
        self.radius = radius
        self.color = QColor(255,0,0)

    def layerType(self):
        return "FooMarker"

    def properties(self):
        return { "radius" : str(self.radius) }

    def startRender(self, context):
        pass

    def stopRender(self, context):
        pass

    def renderPoint(self, point, context):
        # Rendering depends on whether the symbol is selected (QGIS >= 1.5)
        color = context.selectionColor() if context.selected() else self.color
        p = context.renderContext().painter()
        p.setPen(color)
        p.drawEllipse(point, self.radius, self.radius)

    def clone(self):
        return FooSymbolLayer(self.radius)
```

Metoda `layerType()` determină numele stratului simbol, acesta trebuind să fie unic printre toate straturile simbol. Proprietățile sunt utilizate pentru persistența atributelor. Metoda `clone()` trebuie să returneze o copie a stratului simbol, având toate atributele exact la fel. În cele din urmă, mai există metodele de randare: `startRender()` care este apelată înainte de randarea primei entități, și `stopRender()` care oprește randarea. Efectiv, randarea are loc cu ajutorul metodei `renderPoint()`. Coordonatele punctului(punctelor) sunt deja transformate la coordonatele de ieșire.

Pentru polilinii și poligoane singura diferență constă în metoda de randare: ar trebui să utilizați `renderPolyline()` care primește o listă de linii, respectiv `renderPolygon()` care primește lista de puncte de pe inelul exterior ca primul parametru și o listă de inele interioare (sau nici unul), ca al doilea parametru.

De obicei, este convenabilă adăugarea unui GUI pentru setarea atributelor tipului de strat pentru simboluri, pentru a permite utilizatorilor să personalizeze aspectul: în exemplul de mai sus, putem lăsa utilizatorul să seteze raza cercului. Codul de mai jos implementează un astfel de widget

```
class FooSymbolLayerWidget(QgsSymbolLayerV2Widget):
    def __init__(self, parent=None):
        QgsSymbolLayerV2Widget.__init__(self, parent)

        self.layer = None

        # setup a simple UI
        self.label = QLabel("Radius:")
        self.spinRadius = QDoubleSpinBox()
        self.hbox = QHBoxLayout()
        self.hbox.addWidget(self.label)
        self.hbox.addWidget(self.spinRadius)
        self.setLayout(self.hbox)
```

```

self.connect(self.spinRadius, SIGNAL("valueChanged(double)"), \
             self.radiusChanged)

def setSymbolLayer(self, layer):
    if layer.layerType() != "FooMarker":
        return
    self.layer = layer
    self.spinRadius.setValue(layer.radius)

def symbolLayer(self):
    return self.layer

def radiusChanged(self, value):
    self.layer.radius = value
    self.emit(SIGNAL("changed()"))

```

Acest widget poate fi integrat în fereastra de proprietăți a simbolului. În cazul în care tipul de strat simbol este selectat în fereastra de proprietăți a simbolului, se creează o instanță a stratului simbol și o instanță a widget-ului stratului simbol. Apoi, se apelează metoda `setSymbolLayer()` pentru a alocă stratul simbol widget-ului. În acea metodă, widget-ul ar trebui să actualizeze UI pentru a reflecta atributele stratului simbol. Funcția `symbolLayer()` este utilizată la preluarea stratului simbol din fereastra de proprietăți, în scopul folosirii sale pentru simbol.

La fiecare schimbare de atribute, widget-ul ar trebui să emită semnalul `changed()` pentru a permite ferestrei de proprietăți să-i actualizeze previzualizarea simbolului.

Acum mai lipsește doar liantul final: pentru a face QGIS conținent de aceste noi clase. Acest lucru se face prin adăugarea stratului simbol la registru. Este posibilă utilizarea stratului simbol, de asemenea, fără a-l adăuga la registru, dar unele funcționalități nu vor fi disponibile: de exemplu, încărcarea de fiere de proiect cu straturi simbol personalizate sau incapacitatea de a edita atributele stratului în GUI.

Va trebui să creăm metadate pentru stratul simbolului

```

class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

    def __init__(self):
        QgsSymbolLayerV2AbstractMetadata.__init__(self, "FooMarker", QgsSymbolV2.Marker)

    def createSymbolLayer(self, props):
        radius = float(props[QString("radius")]) if QString("radius") in props else 4.0
        return FooSymbolLayer(radius)

    def createSymbolLayerWidget(self):
        return FooSymbolLayerWidget()

```

```
QgsSymbolLayerV2Registry.instance().addSymbolLayerType(FooSymbolLayerMetadata())
```

Ar trebui să transmitem tipul stratului (cel returnat de către strat) și tipul de simbol (marker/linie/umplere) către constructorul clasei părinte. `createSymbolLayer()` are grijă de a crea o instanță de strat simbol cu atributele specificate în dicționarul `props`. (Atenție, tastele reprezintă instanțe `QString`, nu obiecte "str"). Există, de asemenea, metoda `createSymbolLayerWidget()` care returnează setările widget-ului pentru acest tip de strat simbol.

Ultimul pas este de a adăuga acest strat simbol la registru — și am încheiat.

5.9.5 Crearea renderelor Personalizate

Ar putea fi utilă crearea unei noi implementări de render, dacă dorești să personalizezi regulile de selectare a simbolurilor pentru randarea entităților. Unele cazuri de utilizare: simbolul să fie determinat de o combinație de câmpuri, dimensiunea simbolurilor să depindă în funcție de scara curentă, etc

Următorul cod prezintă o simplă randare personalizată, care creează două simboluri de tip marker și apoi alege aleatoriu unul dintre ele pentru fiecare entitate

```
import random

class RandomRenderer(QgsFeatureRendererV2):
    def __init__(self, syms=None):
        QgsFeatureRendererV2.__init__(self, "RandomRenderer")
        self.syms = syms if syms else [QgsSymbolV2.defaultSymbol(Qgis.Point), QgsSymbolV2.defaultSymbol]

    def symbolForFeature(self, feature):
        return random.choice(self.syms)

    def startRender(self, context, vlayer):
        for s in self.syms:
            s.startRender(context)

    def stopRender(self, context):
        for s in self.syms:
            s.stopRender(context)

    def usedAttributes(self):
        return []

    def clone(self):
        return RandomRenderer(self.syms)
```

Constructorul clasei părinte `QgsFeatureRendererV2` are nevoie de numele renderului (trebuie să fie unic printre rendere). Metoda `symbolForFeature()` este cea care decide ce simbol va fi folosit pentru o anumită entitate. `startRender()` și `stopRender()` vor avea grijă de inițializarea/finalizarea randării simbolului. Metoda `usedAttributes()` poate returna o listă de nume de câmpuri a căror prezență a așteaptă renderul. În cele din urmă `clone()` ar trebui să returneze o copie a renderului.

Ca și în cazul straturilor simbol, este posibilă atașarea unui GUI pentru configurarea renderului. Acesta trebuie să fie derivat din `QgsRendererV2Widget`. Următorul exemplu de cod creează un buton care permite utilizatorului setarea primului simbol

```
class RandomRendererWidget(QgsRendererV2Widget):
    def __init__(self, layer, style, renderer):
        QgsRendererV2Widget.__init__(self, layer, style)
        if renderer is None or renderer.type() != "RandomRenderer":
            self.r = RandomRenderer()
        else:
            self.r = renderer
        # setup UI
        self.btn1 = QgsColorButtonV2("Color 1")
        self.btn1.setColor(self.r.syms[0].color())
        self.vbox = QVBoxLayout()
        self.vbox.addWidget(self.btn1)
        self.setLayout(self.vbox)
        self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

    def setColor1(self):
        color = QColorDialog.getColor(self.r.syms[0].color(), self)
        if not color.isValid(): return
        self.r.syms[0].setColor(color)
        self.btn1.setColor(self.r.syms[0].color())

    def renderer(self):
        return self.r
```

Constructorul primește instanțe ale stratului activ (`QgsVectorLayer`), stilul global (`QgsStyleV2`) și renderul curent. Dacă nu există un render sau renderul are alt tip, acesta va fi înlocuit cu noul nostru render, în caz contrar vom folosi renderul curent (care are deja tipul de care avem nevoie). Conținutul widget-ului ar trebui să fie actualizat pentru a arăta starea actuală a renderului. Când dialogul renderului este acceptat, metoda `renderer()` a widgetului este apelată pentru a obține renderul curent — acesta fiind atribuit stratului.

Ultimul bit lipsă este cel al metadatelor renderului i înregistrarea în registru, altfel încărcarea straturilor cu renderul nu va funciona, iar utilizatorul nu va fi capabil să-l selecteze din lista de rendere. Să finalizăm exemplul nostru de RandomRenderer

```
class RandomRendererMetadata(QgsRendererV2AbstractMetadata):
    def __init__(self):
        QgsRendererV2AbstractMetadata.__init__(self, "RandomRenderer", "Random renderer")

    def createRenderer(self, element):
        return RandomRenderer()
    def createRendererWidget(self, layer, style, renderer):
        return RandomRendererWidget(layer, style, renderer)
```

```
QgsRendererV2Registry.instance().addRenderer(RandomRendererMetadata())
```

În mod similar cu straturile simbol, constructorul de metadata abstracte așteaptă numele renderului, nume vizibil pentru utilizatori i numele opional al pictogramei renderului. Metoda `createRenderer()` transmite instanța `QDomElement` care poate fi folosită pentru a restabili starea renderului din arborele DOM. Metoda `createRendererWidget()` creează widget-ul de configurare. Aceasta nu trebuie să fie prezent sau ar putea returna `None`, dacă renderul nu vine cu GUI-ul.

Pentru a asocia o pictogramă renderului ai putea să o asigui în constructorul `QgsRendererV2AbstractMetadata` ca un al treilea argument (opional) — constructorul clasei de bază din funcția `__init__()` a `RandomRendererMetadata` devine

```
QgsRendererV2AbstractMetadata.__init__(self,
    "RandomRenderer",
    "Random renderer",
    QIcon(QPixmap("RandomRendererIcon.png", "png")))
```

Pictograma poate fi asociată ulterior, de asemenea, în orice moment, folosind metoda `setIcon()` a clasei de metadata. Pictograma poate fi încărcată dintr-un fișier (aa cum s-a arătat mai sus), sau dintr-o resursă Qt (PyQt4 include compilatorul `.qrc` pentru Python).

5.10 Lecturi suplimentare

DE EFECTUAT: crearea/modificarea simbolurilor, modificarea stilului (`QgsStyleV2`), modificarea gamelor de culori (`QgsVectorColorRampV2`), rendere bazate pe reguli (citii [această postare pe blog](#)), explorarea straturilor unui simbol i a regitrilor renderelor

Manipularea geometriei

Punctele, liniile și poligoanele, care reprezintă entități spațiale sunt frecvent menționate ca geometrii. În QGIS acestea sunt reprezentate de clasa `QgsGeometry`. Toate tipurile de geometrie posibile sunt frumos prezentate în [pagina de discuții JTS](#).

Uneori, o geometrie poate fi de fapt o colecție de simple geometrii (simple-pări). O astfel de geometrie poartă denumirea de geometrie multi-parte. În cazul în care conține doar un singur tip de geometrie simplă, o denumim multi-punct, multi-linie sau multi-poligon. De exemplu, o ară formată din mai multe insule poate fi reprezentată ca un multi-poligon.

Coordonatele geometriilor pot fi în orice sistem de coordonate de referință (CRS). Când extragem entitățile dintr-un strat, geometriile asociate vor avea coordonatele în CRS-ul stratului.

6.1 Construirea geometriei

Există mai multe opțiuni pentru a crea o geometrie:

- din coordonate

```
gPnt = QgsGeometry.fromPoint(QgsPoint(1,1))
gLine = QgsGeometry.fromPolyline([QgsPoint(1, 1), QgsPoint(2, 2)])
gPolygon = QgsGeometry.fromPolygon([[QgsPoint(1, 1), QgsPoint(2, 2), QgsPoint(2, 1)]])
```

Coordonatele sunt obținute folosind clasa `QgsPoint`.

O polilinie (linie) este reprezentată de o listă de puncte. Poligonul este reprezentat de o listă de inele liniare (de exemplu, linii închise). Primul inel este cel exterior (limita), inele ulterioare opționale reprezentând găurile din poligon.

Geometriile multi-parte merg cu un nivel mai departe: multi-punctele sunt o listă de puncte, multi-liniile o listă de linii iar multi-poligoanele sunt o listă de poligoane.

- din well-known text (WKT)

```
gem = QgsGeometry.fromWkt("POINT(3 4)")
```

- din well-known binary (WKB)

```
g = QgsGeometry()
g.setWkbAndOwnership(wkb, len(wkb))
```

6.2 Accesarea geometriei

În primul rând, ar trebui să găsim tipul geometriei, metoda `wkbType()` fiind cea pe care o puteți utiliza — ea returnând o valoare din enumerarea `Qgis.WkbType`

```
>>> gPnt.wkbType() == QGis.WKBPoint
True
>>> gLine.wkbType() == QGis.WKBLineString
True
>>> gPolygon.wkbType() == QGis.WKBPolygon
True
>>> gPolygon.wkbType() == QGis.WKBMultiPolygon
False
```

Ca alternativă, se poate folosi metoda `type()` care returnează o valoare din enumerarea `QGis.GeometryType`. Există, de asemenea, o funcție ajutătoare `isMultipart()` pentru a afla dacă o geometrie este multipart sau nu.

Pentru a extrage informații din geometrie, există funcțiile accessor pentru fiecare tip de vector. Iată cum le puteți utiliza

```
>>> gPnt.asPoint()
(1, 1)
>>> gLine.asPolyline()
[(1, 1), (2, 2)]
>>> gPolygon.asPolygon()
[[ (1, 1), (2, 2), (2, 1), (1, 1) ]]
```

Notă: tuplurile (x, y) nu reprezintă tupluri reale, ele sunt obiecte `:class:QgsPoint`, valorile fiind accesibile cu ajutorul metodelor `x()` și `y()`.

Pentru geometriile multipart există funcții accessor similare: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Predicate și operațiuni geometrice

QGIS folosește biblioteca GEOS pentru operațiuni geometrice avansate, cum ar fi predicatele geometrice (`contains()`, `intersects()`, ...) și operațiunile de setare (`union()`, `difference()`, ...). Se pot calcula, de asemenea, proprietățile geometrice, cum ar fi suprafața (în cazul poligoanelor) sau lungimea (pentru poligoane și linii)

Iată un mic exemplu care combină iterarea entităților dintr-un strat dat și efectuarea unor calcule bazate pe geometriile lor.

```
# we assume that 'layer' is a polygon layer
features = layer.getFeatures()
for f in features:
    geom = f.geometry()
    print "Area:", geom.area()
    print "Perimeter:", geom.length()
```

Ariile și perimetrele nu iau în considerare CRS-ul atunci când sunt calculate folosind metodele clasei `QgsGeometry`. Pentru un calcul mult mai puternic al ariei și al distanței se poate utiliza clasa `QgsDistanceArea`. În cazul în care proiecțiile sunt dezactivate, calculele vor fi planare, în caz contrar acestea vor fi efectuate pe un elipsoid. Când elipsoidul nu este setat în mod explicit, parametrii WGS84 vor fi utilizați pentru calcule.

```
d = QgsDistanceArea()
d.setProjectionsEnabled(True)

print "distance in meters: ", d.measureLine(QgsPoint(10,10),QgsPoint(11,11))
```

Puteți căuta mai multe exemple de algoritmi care sunt incluși în QGIS și să folosiți aceste metode pentru a analiza și a transforma datele vectoriale. Mai jos sunt prezente câteva trimiteri spre codul unora dintre ele.

Informații suplimentare pot fi găsite în sursele următoare:

- Transformări geometrice: [Reproiectarea algoritmilor](#)
- Aflarea distanței și a ariei folosind clasa `QgsDistanceArea`: [Algoritmul matricei distanțelor](#)
- Algoritmul de transformare din multi-parte în simplă-parte

Proiecii suportate

7.1 Sisteme de coordonate de referință

Sisteme de coordonate de referință (SIR) sunt încapsulate de către clasa `QgsCoordinateReferenceSystem`. Instanțele acestei clase pot fi create prin mai multe moduri diferite:

- specifică CRS-ul după ID-ul său

```
# PostGIS SRID 4326 is allocated for WGS84
crs = QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.PostgisCrsId)
```

QGIS folosește trei ID-uri diferite pentru fiecare sistem de referință:

- `PostgisCrsId` — ID-uri folosite în interiorul bazei de date PostGIS.
- `InternalCrsId` — ID-uri folosite în baza de date QGIS.
- `EpsgCrsId` — ID-uri asignate de către organizația EPSG

În cazul în care nu se specifică altfel în al doilea parametru, PostGIS SRID este utilizat în mod implicit.

- specifică CRS-ul prin well-known text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
      PRIMEM["Greenwich", 0.0], UNIT["degree", 0.017453292519943295], '
      AXIS["Longitude", EAST], AXIS["Latitude", NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
```

- creai un CRS nevalid iar apoi utilizezi una din funcțiile `create*()` pentru a-l inițializa. În următorul exemplu vom folosi irul Proj4 pentru a inițializa proiecția

```
crs = QgsCoordinateReferenceSystem()
crs.createFromProj4("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
```

Este înțelept să verificăm dacă a avut loc crearea cu succes a CRS-ului (de exemplu, efectuând o căutare în baza de date): `isValid()` trebuie să întoarcă `True`.

Rețineți că pentru inițializarea sistemelor de referință spațiale, QGIS trebuie să caute valorile corespunzătoare în baza de date internă `srs.db`. Astfel, în cazul în care creai o aplicație independentă va trebui să stabilești corect căile, cu ajutorul `QgsApplication.setPrefixPath()`, în caz contrar baza de date nu va fi găsită. Dacă executai comenzile din consola QGIS python sau dezvoltai vreun plugin, atunci totul este în regulă: totul este deja configurat pentru dvs.

Accesarea informațiilor sistemului de referință spațial

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
```

```
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in Qgs::units enum)
print "Map units:", crs.mapUnits()
```

7.2 Proiecii

Putei face transformarea între diferitele sisteme de referință spaiale, cu ajutorul clasei `QgsCoordinateTransform`. Cel mai simplu mod de a o folosi este de a crea CRS-urile sursă și destinație și să construiești cu ele o instanță `QgsCoordinateTransform`. Apoi, doar repetai apelul funcției `transform()` pentru a realiza transformarea. În mod implicit, aceasta face transformarea în ordinea deja precizată, dar este capabilă de a face și transformarea inversă

```
crsSrc = QgsCoordinateReferenceSystem(4326) # WGS 84
crsDest = QgsCoordinateReferenceSystem(32633) # WGS 84 / UTM zone 33N
xform = QgsCoordinateTransform(crsSrc, crsDest)

# forward transformation: src -> dest
pt1 = xform.transform(QgsPoint(18,5))
print "Transformed point:", pt1

# inverse transformation: dest -> src
pt2 = xform.transform(pt1, QgsCoordinateTransform.ReverseTransform)
print "Transformed back:", pt2
```

Folosirea suportului de hartă

Widget-ul suportului de hartă este, probabil, cel mai important în QGIS, deoarece prezintă o hartă compusă din straturi suprapuse și permite atât interacțiunea cu harta cât și cu straturile. Suportul arată întotdeauna o parte a hărții definite de caseta de încadrare curentă. Interacțiunea se realizează prin utilizarea unor **instrumente pentru hartă**: există instrumente de panoramare, de mărire, de identificare a straturilor, de măsurare, de editare vectorială și altele. Similar altor programe de grafică, există întotdeauna un instrument activ, iar utilizatorul poate comuta între instrumentele disponibile.

Suportul hărții este implementat ca și clasa `QgsMapCanvas`, în modulul `qgis.gui`. Implementarea se bazează pe cadrul de lucru `Qt Graphics View`. Acest cadru, în general, pune la dispoziție o suprafață și o fereastră de vizualizare a acesteia, unde sunt plasate elementele grafice personalizate, utilizatorul putând interacționa cu ele. Vom presupune că v-ai familiarizat suficient cu `Qt`, pentru a înțelege conceptele de scenă grafică, vizualizare și elemente. Dacă nu, vă rugăm să citiți [o prezentare generală a cadrului de lucru](#).

Ori de câte ori harta a fost deplasată, mărită/micorată (sau alte acțiuni care declanșează o recitire), harta este randată iarăși în interiorul granielor curente. Straturile sunt transformate într-o imagine (folosind clasa `QgsMapRenderer`) iar acea imagine este afișată pe suport. Elementul grafic (în termeni ai cadrului de lucru `Qt Graphics View`) responsabil pentru afișarea hărții este `QgsMapCanvasMap`. Această clasă controlează, de asemenea, recitirea hărții randate. În afară de acest element, care acționează ca fundal, pot exista mai multe **elemente ale suportului hărții**. Elementele tipice suportului de hartă sunt benzile elastice (utilizate pentru măsurare, editare vectorială etc) sau marcasele nodurilor. Elementele suportului sunt de obicei utilizate pentru a oferi un răspuns vizual pentru instrumentele hărții, de exemplu, atunci când se creează un nou poligon, instrumentul corepunzător creează o bandă elastică de forma actuală a poligonului. Toate elementele suportului de hartă reprezintă subclase ale `QgsMapCanvasItem` care adaugă mai multe funcționalități obiectelor de bază `QGraphicsItem`.

Pentru a rezuma, arhitectura suportului pentru hartă constă în trei concepte:

- suportul de hartă — pentru vizualizarea hărții
- elementele — elemente suplimentare care pot fi afișate în suportul hărții
- instrumentele hărții — pentru interacțiunea cu suportul hărții

8.1 Încapsularea suportului de hartă

Canevasul hărții este un widget ca orice alt widget `Qt`, aa că utilizarea este la fel de simplă ca și crearea și afișarea lui

```
canvas = QgsMapCanvas()
canvas.show()
```

Acest cod va produce o fereastră de sine stătătoare cu suport pentru hartă. Ea poate fi, de asemenea, încorporată într-un widget sau într-o fereastră deja existentă. Atunci când se utilizează fiere `.ui` în `Qt Designer`, puneți un `QWidget` pe formă pe care, ulterior, o veți promova la o nouă clasă: setați `QgsMapCanvas` ca nume de clasă și stabiliți `qgis.gui` ca fiier antet. Utilitarul `pyuic4` va avea grijă de ea. Acesta este un mod foarte convenabil de încapsulare a suportului. Cealaltă posibilitate este de a scrie manual codul pentru a construi suportul hărții și alte widget-uri (în calitate de copii ai ferestrei principale sau de dialog), apoi creai o aezare în pagină.

În mod implicit, canevasul hărții are un fundal negru și nu utilizează anti-zimare. Pentru a seta fundalul alb și pentru a permite anti-zimare pentru o redare mai bună

```
canvas.setCanvasColor(Qt.white)
canvas.enableAntiAliasing(True)
```

(În cazul în care vă întrebați, Qt vine de la modulul PyQt4.QtCore iar Qt.white este una dintre instanele QColor predefinite.)

Acum este timpul adăugării mai multor straturi de hartă. Vom deschide mai întâi un strat și-l vom adăuga la registrul straturilor. Apoi vom stabili extinderea canevasului și vom stabili lista straturilor

```
layer = QgsVectorLayer(path, name, provider)
if not layer.isValid():
    raise IOError, "Failed to open the layer"

# add layer to the registry
QgsMapLayerRegistry.instance().addMapLayer(layer)

# set extent to the extent of our layer
canvas.setExtent(layer.extent())

# set the map canvas layer set
canvas.setLayerSet([QgsMapCanvasLayer(layer)])
```

După executarea acestor comenzi, suportul ar trebui să arate stratul pe care le-ai încărcat.

8.2 Folosirea instrumentelor în suportul de hartă

Următorul exemplu construiește o fereastră care conține un canevas și instrumente de bază pentru panoramare și mărire hartă. Acțiunile sunt create pentru activarea fiecărui instrument: panoramarea se face cu QgsMapToolPan, mărirea/micorarea cu o pereche de instane a QgsMapToolZoom. Acțiunile sunt setate ca selectabile, și asignate ulterior instrumentelor, pentru a permite gestionarea automată a stării selectabile a acțiunilor - atunci când un instrument al hărții este activat, acțiunea sa este marcată ca fiind selectată iar acțiunea instrumentului anterior este deselectată. Instrumentele sunt activate folosindu-se metoda setMapTool().

```
from qgis.gui import *
from PyQt4.QtGui import QAction, QMainWindow
from PyQt4.QtCore import SIGNAL, Qt, QString

class MyWnd(QMainWindow):
    def __init__(self, layer):
        QMainWindow.__init__(self)

        self.canvas = QgsMapCanvas()
        self.canvas.setCanvasColor(Qt.white)

        self.canvas.setExtent(layer.extent())
        self.canvas.setLayerSet([QgsMapCanvasLayer(layer)])

        self.setCentralWidget(self.canvas)

        actionZoomIn = QAction(QString("Zoom in"), self)
        actionZoomOut = QAction(QString("Zoom out"), self)
        actionPan = QAction(QString("Pan"), self)

        actionZoomIn.setCheckable(True)
        actionZoomOut.setCheckable(True)
        actionPan.setCheckable(True)

        self.connect(actionZoomIn, SIGNAL("triggered()"), self.zoomIn)
```

```

self.connect(actionZoomOut, SIGNAL("triggered()"), self.zoomOut)
self.connect(actionPan, SIGNAL("triggered()"), self.pan)

self.toolbar = self.addToolBar("Canvas actions")
self.toolbar.addAction(actionZoomIn)
self.toolbar.addAction(actionZoomOut)
self.toolbar.addAction(actionPan)

# create the map tools
self.toolPan = QgsMapToolPan(self.canvas)
self.toolPan.setAction(actionPan)
self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
self.toolZoomIn.setAction(actionZoomIn)
self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
self.toolZoomOut.setAction(actionZoomOut)

self.pan()

def zoomIn(self):
    self.canvas.setMapTool(self.toolZoomIn)

def zoomOut(self):
    self.canvas.setMapTool(self.toolZoomOut)

def pan(self):
    self.canvas.setMapTool(self.toolPan)

```

Putei pune codul de mai sus într-un fiier, de exemplu, `mywnd.py` i să-l încerci apoi în consola Python din QGIS. Acest cod va pune stratul curent selectat în noul canvas creat

```

import mywnd
w = mywnd.MyWnd(qgis.utils.iface.activeLayer())
w.show()

```

Doar asigurai-vă că fiierul `mywnd.py` se află în calea de căutare pentru Python (`sys.path`). În cazul în care nu este, puteți pur și simplu să o adăugați: `sys.path.insert(0, '/calea/mea')` — altfel declarația de import nu va reuși, negăsind modulul.

8.3 Benzile elastice i marcajele nodurilor

Pentru a arăta unele date suplimentare în partea de sus a hărții, folosii elemente ale canvasului. Cu toate că este posibil să se creeze clase de elemente de canvas personalizate (detaliat mai jos), există două clase de elemente confortabile `QgsRubberBand` pentru desenarea de polilinii sau poligoane, i `QgsVertexMarker` pentru puncte. Amândouă lucrează cu coordonatele hărții, astfel încât o formă este mutată/scalată în mod automat atunci când canvasul este rotit sau mărit.

Pentru a afia o polilinie

```

r = QgsRubberBand(canvas, False) # False = not a polygon
points = [QgsPoint(-1, -1), QgsPoint(0, 1), QgsPoint(1, -1)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)

```

Pentru a afia un poligon

```

r = QgsRubberBand(canvas, True) # True = a polygon
points = [[QgsPoint(-1, -1), QgsPoint(0, 1), QgsPoint(1, -1)]]
r.setToGeometry(QgsGeometry.fromPolygon(points), None)

```

Reinei că punctele pentru poligon nu reprezintă o simplă listă: în fapt, aceasta este o listă de inele conținând inele liniare ale poligonului: primul inel reprezintă grania exterioară, în plus (opional) inelele corespund găurilor din poligon.

Benzile elastice acceptă unele personalizări, i anume schimbarea culorii i a lăimii liniei

```
r.setColor(QColor(0, 0, 255))
r.setWidth(3)
```

Elementele suportului sunt legate de suportul hării. Pentru a le ascunde temporar (i a le arăta din nou, folosii combinaia `hide()` i `show()`. Pentru a elimina complet elementul, trebuie să-l eliminăm de pe scena canvasului

```
canvas.scene().removeItem(r)
```

(În C++ este posibilă tergerea doar a elementului, însă în Python `del r` ar terge doar referința iar obiectul va exista în continuare, acesta fiind deinut de suport)

Banda elastică poate fi de asemenea utilizată pentru desenarea de puncte, însă, clasa `QgsVertexMarker` este mai potrivită pentru aceasta (`QgsRubberBand` ar trasa doar un dreptunghi în jurul punctului dorit). Cum să utilizai simbolul nodului

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPoint(0, 0))
```

În acest mod se va desena o cruciuliă roie pe poziția [0,0]. Este posibilă personalizarea tipului pictogramei, dimensiunea, culoarea i lăimea instrumentului de desenare

```
m.setColor(QColor(0, 255, 0))
m.setIconSize(5)
m.setIconType(QgsVertexMarker.ICON_BOX) # or ICON_CROSS, ICON_X
m.setPenWidth(3)
```

Pentru ascunderea temporară a markerilor vertex i pentru eliminarea lor de pe suport, același lucru este valabil i pentru benzile elastice.

8.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă

Putei crea propriile instrumente, pentru a implementa un comportament personalizat pentru acțiunile executate de către utilizatori pe canvas.

Instrumentele de hartă ar trebui să motenească clasa `QgsMapTool` sau orice altă clasă derivată, i să fie selectate ca instrumente active pe suport, folosindu-se metoda `setMapTool()`, aa cum am văzut deja.

Iată un exemplu de instrument pentru hartă, care permite definirea unei limite dreptunghiulare, făcând clic i trăgând cursorul mouse-ului pe canvas. După ce este definit dreptunghiul, coordonatele sale sunt afișate în consolă. Se utilizează elementele benzii elastice descrise mai înainte, pentru a arăta dreptunghiul selectat, aa cum a fost definit.

```
class RectangleMapTool(QgsMapToolEmitPoint):
    def __init__(self, canvas):
        self.canvas = canvas
        QgsMapToolEmitPoint.__init__(self, self.canvas)
        self.rubberBand = QgsRubberBand(self.canvas, Qgs.Polygon)
        self.rubberBand.setColor(Qt.red)
        self.rubberBand.setWidth(1)
        self.reset()

    def reset(self):
        self.startPoint = self.endPoint = None
        self.isEmittingPoint = False
        self.rubberBand.reset(Qgs.Polygon)

    def canvasPressEvent(self, e):
        self.startPoint = self.toMapCoordinates(e.pos())
        self.endPoint = self.startPoint
        self.isEmittingPoint = True
```

```

self.showRect(self.startPoint, self.endPoint)

def canvasReleaseEvent(self, e):
    self.isEmittingPoint = False
    r = self.rectangle()
    if r is not None:
        print "Rectangle:", r.xMinimum(), r.yMinimum(), r.xMaximum(), r.yMaximum()

def canvasMoveEvent(self, e):
    if not self.isEmittingPoint:
        return

    self.endPoint = self.toMapCoordinates(e.pos())
    self.showRect(self.startPoint, self.endPoint)

def showRect(self, startPoint, endPoint):
    self.rubberBand.reset(QGis.Polygon)
    if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
        return

    point1 = QgsPoint(startPoint.x(), startPoint.y())
    point2 = QgsPoint(startPoint.x(), endPoint.y())
    point3 = QgsPoint(endPoint.x(), endPoint.y())
    point4 = QgsPoint(endPoint.x(), startPoint.y())

    self.rubberBand.addPoint(point1, False)
    self.rubberBand.addPoint(point2, False)
    self.rubberBand.addPoint(point3, False)
    self.rubberBand.addPoint(point4, True)    # true to update canvas
    self.rubberBand.show()

def rectangle(self):
    if self.startPoint is None or self.endPoint is None:
        return None
    elif self.startPoint.x() == self.endPoint.x() or self.startPoint.y() == self.endPoint.y():
        return None

    return QgsRectangle(self.startPoint, self.endPoint)

def deactivate(self):
    QgsMapTool.deactivate(self)
    self.emit(SIGNAL("deactivated()"))

```

8.5 Dezvoltarea elementelor personalizate pentru suportul de hartă

DE EFECTUAT: how to create a map canvas item

```

import sys
from qgis.core import QgsApplication
from qgis.gui import QgsMapCanvas

def init():
    a = QgsApplication(sys.argv, True)
    QgsApplication.setPrefixPath('/home/martin/qgis/inst', True)
    QgsApplication.initQgis()
    return a

def show_canvas(app):
    canvas = QgsMapCanvas()

```

```
canvas.show()
app.exec_()
app = init()
show_canvas(app)
```

Randarea hărilor i imprimarea

Există, în general, două abordări atunci când datele de intrare ar trebui să fie randate într-o hartă: fie o modalitate rapidă, folosind `QgsMapRenderer`, fie producerea unei ieiri mai rafinate, prin compunerea hărții cu ajutorul clasei `QgsComposition`.

9.1 Randarea simplă

Randai mai multe straturi, folosind `QgsMapRenderer` — creai destinaia dispozitivului de colorare (`QImage`, `QPainter` etc), setai stratul, limitele sale, dimensiunea de ieire i efectuai randarea

```
# create image
img = QImage(QSize(800, 600), QImage.Format_ARGB32_Premultiplied)

# set image's background color
color = QColor(255, 255, 255)
img.fill(color.rgb())

# create painter
p = QPainter()
p.begin(img)
p.setRenderHint(QPainter.Antialiasing)

render = QgsMapRenderer()

# set layer set
lst = [layer.getLayerID()] # add ID of every layer
render.setLayerSet(lst)

# set extent
rect = QgsRect(render.fullExtent())
rect.scale(1.1)
render.setExtent(rect)

# set output size
render.setOutputSize(img.size(), img.logicalDpiX())

# do the rendering
render.render(p)

p.end()

# save image
img.save("render.png", "png")
```

9.2 Randarea straturilor cu diferite CRS-uri

Dacă avei mai mult de un singur strat, iar dacă acestea au CRS-uri diferite, exemplul simplu de mai sus nu va funciona: pentru a obține valorile corecte din extinderile calculate, va trebui să setai în mod explicit CRS-ul destinației și să activezi reproiectarea OTF ca în exemplul de mai jos (numai partea de configurare a randării se raportează)

```
...
# set layer set
layers = QgsMapLayerRegistry.instance().mapLayers()
lst = layers.keys()
render.setLayerSet(lst)

# Set destination CRS to match the CRS of the first layer
render.setDestinationCrs(layers.values()[0].crs())
# Enable OTF reprojection
render.setProjectionsEnabled(True)
...
```

9.3 Generarea folosind Compozitorul de hări

Compozitorul de hări reprezintă un instrument foarte util în cazul în care dorești să elaborezi ceva mai sofisticat decât simpla randare de mai sus. Utilizând Constructorului este posibilă crearea unor machete complexe de hări, conținând extrase de hartă, etichete, legendă, tabele și alte elemente care sunt de obicei prezente pe hărțile tipărite. Machetele pot fi apoi exportate în format PDF, ca imagini raster sau pot fi transmise direct la o imprimantă.

The composer consists of a bunch of classes. They all belong to the core library. QGIS application has a convenient GUI for placement of the elements, though it is not available in the GUI library. If you are not familiar with [Qt Graphics View framework](#), then you are encouraged to check the documentation now, because the composer is based on it.

Clasa centrală a Compozitorului este `QgsComposition`, care este derivată din `QGraphicsScene`. Să creăm una

```
mapRenderer = iface.mapCanvas().mapRenderer()
c = QgsComposition(mapRenderer)
c.setPlotStyle(QgsComposition.Print)
```

Reinei: compoziția este o instanță a `QgsMapRenderer`. În cod, ne așteptăm să rulăm în interiorul aplicației QGIS și, astfel, să folosim render-ul suportului de hartă. Compoziția utilizează diverși parametri ai render-ului, cei mai importanți fiind setul implicit de straturi de hartă și granițele curente. Atunci când utilizezi compozitorul într-o aplicație independentă, vă puteți crea propria dvs. instanță de render de hări, în același mod cum s-a arătat în secțiunea de mai sus, și să-l transmiteți compoziției.

Este posibilă adăugarea diferitelor elemente (hartă, etichete, ...) în compoziție — aceste elemente trebuie să fie descendenți ai clasei `QgsComposerItem`. Elementele suportate în prezent sunt:

- harta — acest element indică bibliotecilor unde să pună harta. Vom crea o hartă și o vom întinde peste întreaga dimensiune a hârtiei

```
x, y = 0, 0
w, h = c.paperWidth(), c.paperHeight()
composerMap = QgsComposerMap(c, x, y, w, h)
c.addItem(composerMap)
```

- eticheta — permite afișarea textelor. Este posibilă modificarea fontului, culoarea, alinierea și marginea

```
composerLabel = QgsComposerLabel(c)
composerLabel.setText("Hello world")
composerLabel.adjustSizeToText()
c.addItem(composerLabel)
```


- legenda

```
legend = QgsComposerLegend(c)
legend.model().setLayerSet(mapRenderer.layerSet())
c.addItem(legend)
```

- scara grafică

```
item = QgsComposerScaleBar(c)
item.setStyle('Numeric') # optionally modify the style
item.setComposerMap(composerMap)
item.applyDefaultSize()
c.addItem(item)
```

- săgeată
- imagine
- formă
- tabelă

În mod implicit, elementele compozitorului nou creat au poziia zero (colul din stânga sus a paginii) i dimensiunea zero. Poziia i dimensiunea sunt măsurate întotdeauna în milimetri

```
# set label 1cm from the top and 2cm from the left of the page
composerLabel.setItemPosition(20, 10)
# set both label's position and size (width 10cm, height 3cm)
composerLabel.setItemPosition(20, 10, 100, 30)
```

În jurul fiecărui element este desenat, în mod implicit, un cadru. Astfel se elimină cadrul

```
composerLabel.setFrame(False)
```

Pe lângă crearea manuală a elementele compozitorului, QGIS are suport pentru abloane, care sunt, în esenă, compoziții cu toate elementele lor salvate într-un fiier .qpt (cu sintaxă XML). Din păcate, această funcționalitate nu este încă disponibilă în API.

Odată ce compoziția este gata (elementele compozitorului au fost create i adăugate la compoziție), putem trece la producerea unui raster i/sau a unei ieiri vectoriale.

Setările de ieire implicite pentru compoziție sunt pentru o pagină A4 i o rezoluție de 300 DPI. Le puteți modifica, atunci când este necesar. Dimensiunea hârtiei este specificată în milimetri

```
c.setPaperSize(width, height)
c.setPrintResolution(dpi)
```

9.3.1 Ieire ca imagine raster

Următorul fragment de cod arată cum se randează o compoziție într-o imagine raster

```
dpi = c.printResolution()
dpmm = dpi / 25.4
width = int(dpmm * c.paperWidth())
height = int(dpmm * c.paperHeight())

# create output image and initialize it
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

# render the composition
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
```

```
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()

image.save("out.png", "png")
```

9.3.2 leire în format PDF

Următorul fragment de cod randează o compoziție într-un fișier PDF

```
printer = QPrinter()
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName("out.pdf")
printer.setPaperSize(QSizeF(c.paperWidth(), c.paperHeight()), QPrinter.Millimeter)
printer.setFullPage(True)
printer.setColorMode(QPrinter.Color)
printer.setResolution(c.printResolution())

pdfPainter = QPainter(printer)
paperRectMM = printer.pageRect(QPrinter.Millimeter)
paperRectPixel = printer.pageRect(QPrinter.DevicePixel)
c.render(pdfPainter, paperRectPixel, paperRectMM)
pdfPainter.end()
```

Expresii, filtrarea și calculul valorilor

QGIS has some support for parsing of SQL-like expressions. Only a small subset of SQL syntax is supported. The expressions can be evaluated either as boolean predicates (returning True or False) or as functions (returning a scalar value).

Trei tipuri de bază sunt acceptate:

- — număr atât numere întregi cât și numere zecimale, de exemplu, 123, 3.14
- ir — acesta trebuie să fie cuprins între ghilimele simple: 'hello world'
- referință către coloană — atunci când se evaluează, referința este substituită cu valoarea reală a câmpului. Numele nu sunt protejate.

Următoarele operațiuni sunt disponibile:

- operatori aritmetici: +, -, *, /, ^
- paranteze: pentru forarea priorității operatorului: (1 + 1) * 3
- plus și minus unari: -12, +5
- funcții matematice: sqrt, sin, cos, tan, asin, acos, atan
- funcții geometrice: \$area, \$length
- conversion functions: to int, to real, to string

și următoarele predicate sunt suportate:

- comparație: =, !=, >, >=, <, <=
- potrivirea paternurilor: LIKE (folosind % și _), ~ (expresii regulate)
- predicate logice: AND, OR, NOT
- verificarea valorii NULL: IS NULL, IS NOT NULL

Exemple de predicate:

- 1 + 2 = 3
- sin(angle) > 0
- 'Hello' LIKE 'He%'
- (x > 10 AND y > 10) OR z = 0

Exemple de expresii scalare:

- 2 ^ 10
- sqrt(val)
- \$length + 1

10.1 Parsarea expresiilor

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> exp.hasParserError()
False
>>> exp = QgsExpression('1 + 1 = ')
>>> exp.hasParserError()
True
>>> exp.parserErrorString()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

10.2 Evaluarea expresiilor

10.2.1 Expresii de bază

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> value = exp.evaluate()
>>> value
1
```

10.2.2 Expresii cu entități

Următorul exemplu va evalua expresia dată față de o entitate. “Column” este numele câmpului din strat.

```
>>> exp = QgsExpression('Column = 99')
>>> value = exp.evaluate(feature, layer.pendingFields())
>>> bool(value)
True
```

De asemenea, puteți folosi `QgsExpression.prepare()`, dacă trebuie să verificați mai mult de o entitate. Utilizarea `QgsExpression.prepare()` va spori viteza evaluării.

```
>>> exp = QgsExpression('Column = 99')
>>> exp.prepare(layer.pendingFields())
>>> value = exp.evaluate(feature)
>>> bool(value)
True
```

10.2.3 Tratarea erorilor

```
exp = QgsExpression("1 + 1 = 2 ")
if exp.hasParserError():
    raise Exception(exp.parserErrorString())

value = exp.evaluate()
if exp.hasEvalError():
    raise ValueError(exp.evalErrorString())

print value
```

10.3 Exemple

Următorul exemplu poate fi folosit pentru a filtra un strat și pentru a întoarce orice entitate care se potrivește unui predicat.

```
def where(layer, exp):
    print "Where"
    exp = QgsExpression(exp)
    if exp.hasParserError():
        raise Exception(exp.parserErrorString())
    exp.prepare(layer.pendingFields())
    for feature in layer.getFeatures():
        value = exp.evaluate(feature)
        if exp.hasEvalError():
            raise ValueError(exp.evalErrorString())
        if bool(value):
            yield feature

layer = qgis.utils.iface.activeLayer()
for f in where(layer, 'Test > 1.0'):
    print f + " Matches expression"
```

Citirea i stocarea setărilor

De multe ori, pentru un plugin, este utilă salvarea unor variabile, astfel încât utilizatorul să nu trebuiască să le reintroducă sau să le reselecteze, la fiecare rulare a plugin-ului.

Aceste variabile pot fi salvate cu ajutorul Qt i QGIS API. Pentru fiecare variabilă ar trebui să alegeți o cheie care va fi folosită pentru a accesa variabila — pentru culoarea preferată a utilizatorului ai putea folosi o cheie de genul “culoare_favorită” sau orice alt ir semnificativ. Este recomandabil să folosiți o oarecare logică în denumirea cheilor.

Putem face diferența între mai multe tipuri de setări:

- **setări globale** — acestea in de utilizatorul unei anumite maini. QGIS însui stochează o mulime de setări globale, cum ar fi, de exemplu, dimensiunea ferestrei principale sau toleranța implicită pentru acroare. Această funcționalitate este furnizată direct de cadrul de lucru Qt, prin intermediul clasei `QSettings`. În mod implicit, această clasă îi depozitează setările în modul “nativ” al sistemului dvs, care este — în registru (pentru Windows), în fiierul `.plist` (pentru Mac OS X) sau în fiierul `.ini` (pentru Unix). [Documentația QSettings](#) este cuprinzătoare, aa că vă vom prezenta doar un simplu exemplu

```
def store():
    s = QSettings()
    s.setValue("myplugin/mytext", "hello world")
    s.setValue("myplugin/myint", 10)
    s.setValue("myplugin/myreal", 3.14)

def read():
    s = QSettings()
    mytext = s.value("myplugin/mytext", "default text")
    myint = s.value("myplugin/myint", 123)
    myreal = s.value("myplugin/myreal", 2.71)
```

Al doilea parametru al metodei `value()` este opțional i specifică valoarea implicită, dacă nu există nici o valoare anterioară stabilită pentru setare.

- **setările proiectului** — variază între diferite proiecte i, prin urmare, ele sunt conectate cu un fiier de proiect. Culoarea de fundal a suportului hărții sau sistemul de coordonate de referință (CRS), de exemplu — fundal alb i WGS84 ar putea fi potrivite pentru un anumit proiect, în timp ce fondul galben i proiecția UTM ar putea fi mai bune pentru altul. În continuare este dat un exemplu de utilizare

```
proj = QgsProject.instance()

# store values
proj.writeEntry("myplugin", "mytext", "hello world")
proj.writeEntry("myplugin", "myint", 10)
proj.writeEntry("myplugin", "mydouble", 0.01)
proj.writeEntry("myplugin", "mybool", True)

# read values
mytext = proj.readEntry("myplugin", "mytext", "default text")[0]
myint = proj.readNumEntry("myplugin", "myint", 123)[0]
```

După cum putei vedea, metoda `writeEntry()` este folosită pentru toate tipurile de date, dar există mai multe metode pentru a seta înapoi setarea, iar cea corespunzătoare trebuie să fie selectată pentru fiecare tip de date.

- **setările stratului hării** — aceste setări sunt legate de o anumită instanță a unui strat de hartă cu un proiect. Acestea *nu* sunt conectate cu sursa de date a stratului, aa că dacă vei crea două instane ale unui strat de hartă dintr-un fiier shape, ele nu vor partaja setările. Setările sunt stocate în fiierul proiectului, astfel încât, în cazul în care utilizatorul deschide iarăi proiectul, setările legate de strat vor fi din nou acolo. Această funcționalitate a fost adăugată în QGIS v1.4. API-ul este similar cu `QSettings` — luând și returnând instane `QVariant`

```
# save a value
layer.setCustomProperty("mytext", "hello world")

# read the value again
mytext = layer.customProperty("mytext", "default text")
```

Comunicarea cu utilizatorul

Această secțiune prezintă câteva metode și elemente care ar trebui să fie utilizate pentru a comunica cu utilizatorul, în scopul meninerii coerenței interfeței cu utilizatorul.

12.1 Afiarea mesajelor. Clasa `QgsMessageBar`

Folosirea casetelor de mesaje poate fi o idee rea, din punctul de vedere al experienței utilizatorului. Pentru a arăta o mică linie de informații sau un mesaj de avertizare/eroare, bara QGIS de mesaje este, de obicei, o opțiune mai bună.

Folosind referința către obiectul interfeței QGIS, puteți afișa un text în bara de mesaje, cu ajutorul următorului cod

```
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that", level=QgsMe
```

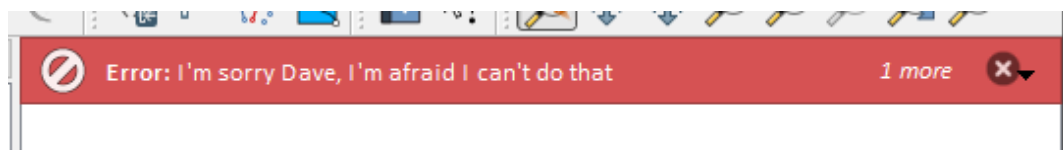


Figure 12.1: Bara de mesaje a QGIS

Puteți seta o durată, pentru afișarea pentru o perioadă limitată de timp

```
iface.messageBar().pushMessage("Error", "'Oops, the plugin is not working as it should", level=Q
```

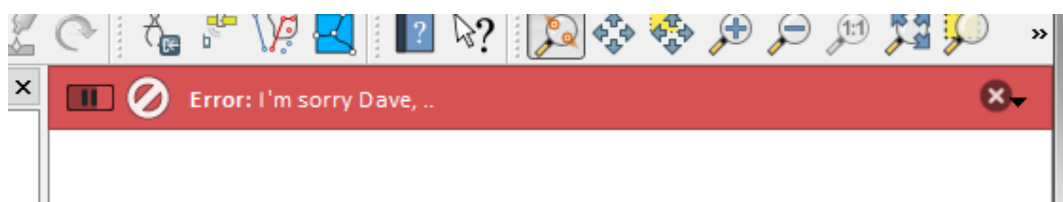


Figure 12.2: Bara de mesaje a QGIS, cu cronometru

Exemplele de mai sus arată o bară de eroare, dar parametrul `level` poate fi utilizat pentru a crea mesaje de avertizare sau informative, folosind constantele `QgsMessageBar.WARNING` și respectiv `QgsMessageBar.INFO`.

Widget-urile pot fi adăugate la bara de mesaje, cum ar fi, de exemplu, un buton pentru afișarea mai multor informații

```
def showError():
    pass
```

```
widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
```

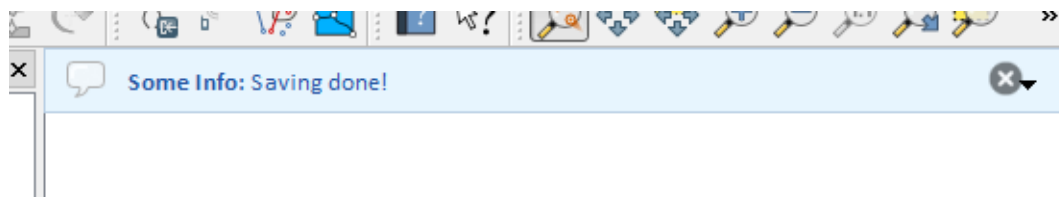


Figure 12.3: Bara de mesaje a QGIS (info)

```
button = QPushButton(widget)
button.setText("Show Me")
button.pressed.connect(showError)
widget.layout().addWidget(button)
iface.messageBar().pushWidget(widget, QgsMessageBar.WARNING)
```

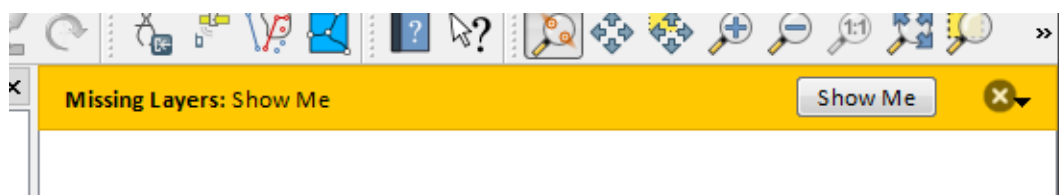


Figure 12.4: Bara de mesaje a QGIS, cu un buton

Putei utiliza o bară de mesaje chiar și în propria fereastră de dialog, în loc să apelei la o casetă de text, sau să arătați mesajul în fereastra principală a QGIS

```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy(QSizePolicy.Minimum, QSizePolicy.Fixed)
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0, 0, 0, 0)
        self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
        self.buttonbox.accepted.connect(self.run)
        self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
        self.layout().addWidget(self.bar, 0, 0, 1, 1)

    def run(self):
        self.bar.pushMessage("Hello", "World", level=QgsMessageBar.INFO)
```

12.2 Afiarea progresului

Barele de progres pot fi, de asemenea, incluse în bara de mesaje QGIS, din moment ce, aa cum am văzut, aceasta acceptă widget-uri. Iată un exemplu pe care îl puteți încerca în consolă.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
```

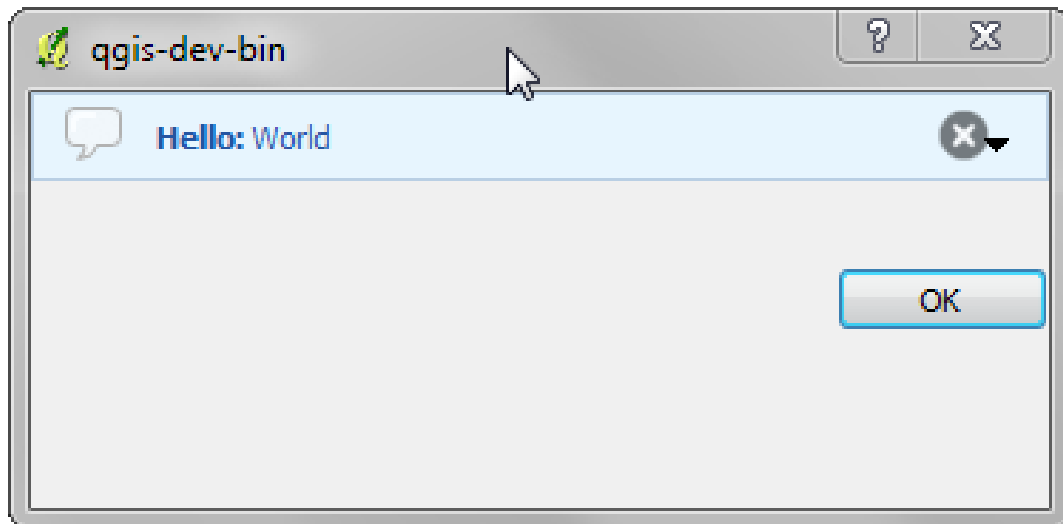


Figure 12.5: Bara de mesaje a QGIS, într-o fereastră de dialog

```
progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

De asemenea, aveți posibilitatea să utilizați bara de stare internă pentru a raporta progresul, ca în exemplul următor

```
count = layers.featureCount()
for i, feature in enumerate(features):
    #do something time-consuming here
    ...
    percent = i / float(count) * 100
    iface.mainWindow().statusBar().showMessage("Processed {} %".format(int(percent)))
iface.mainWindow().statusBar().clearMessage()
```

12.3 Jurnalizare

Puteți utiliza sistemul de jurnalizare al QGIS, pentru a salva toate informațiile pe care doriți să le înregistrați, cu privire la execuția codului dvs.

```
# You can optionally pass a 'tag' and a 'level' parameters
QgsMessageLog.logMessage("Your plugin code has been executed correctly", 'MyPlugin', QgsMessageLog.INFO)
QgsMessageLog.logMessage("Your plugin code might have some problems", level=QgsMessageLog.WARNING)
QgsMessageLog.logMessage("Your plugin code has crashed!", level=QgsMessageLog.CRITICAL)
```

Dezvoltarea plugin-urilor Python

Este posibil să se creeze plugin-uri în limbajul de programare Python. În comparație cu plugin-urile clasice scrise în C++ acestea ar trebui să fie mai ușor de scris, de înțeles, de menținut și de distribuit, din cauza naturii dinamice a limbajului Python.

Plugin-urile Python sunt listate, împreună cu plugin-urile C++, în managerul de plugin-uri QGIS. Ele sunt căutate în aceste căi:

- UNIX/Mac: `~/ .qgis/python/plugins` and `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/ .qgis/python/plugins` and `(qgis_prefix)/python/plugins`

Directorul de casă (notat `~`) pe Windows este, de obicei, ceva de genul `C:\Documents and Settings\ (user)` (în Windows XP sau versiunile anterioare) sau `C:\Users\ (user)`. Deoarece QGIS utilizează Python 2.7, subdirectoarele acestor căi trebuie să conțină un fișier `__init__.py`, pentru a fi considerate pachete Python care pot fi importate ca plugin-uri.

Note: Prin ataarea `QGIS_PLUGINPATH` căii unui director existent, puteți vedea această cale în lista căilor de căutare pentru plugin-uri.

Pai:

1. *Ideea:* Conturată o idee despre ceea ce vrei să faci cu noul plugin QGIS. De ce-l faci? Ce problemă dorești să rezolvi? Există deja un alt plugin pentru această problemă?
2. *Creare fișiere:* Se creează fișierele descrise în continuare. Un punct de plecare (`__init__.py`). Completai *!Metadatele plugin-ului* (`metadata.txt`). Un corp python principal al plugin-ului (`mainplugin.py`). O formă în QT-Designer (`form.ui`), cu al său `resources.qrc`.
3. *Scris codul:* Scrieți codul în interiorul `mainplugin.py`
4. *Testul:* Închideți și re-deschideți QGIS, apoi importați-l din nou. Verificați dacă totul este în regulă.
5. *Publicare:* Se publică plugin-ul în depozitul QGIS sau vă faceți propriul depozit ca un “arsenal” de “arme GIS” personale

13.1 Scrierea unui plugin

De la introducerea plugin-urilor Python în QGIS, a apărut un număr de plugin-uri - pe [pagina wiki a Depozitelor de Plugin-uri](#) puteți găsi unele dintre ele, le puteți utiliza sursa pentru a afla mai multe despre programarea în PyQGIS sau să aflați dacă nu cumva duplicați efortul de dezvoltare. Echipa QGIS menține, de asemenea, un *Depozitul oficial al plugin-urilor python*. Sunteți gata de a crea un plugin, dar nu aveți nici o idee despre cum ai putea începe? În [pagina wiki](#) cu idei de plugin-uri Python sunt listate dorințele comunității!

13.1.1 Fiierle Plugin-ului

Iată structura de directoare a exemplului nostru de plugin

```
PYTHON_PLUGINS_PATH/  
MyPlugin/  
  __init__.py    --> *required*  
  mainPlugin.py  --> *required*  
  metadata.txt   --> *required*  
  resources.qrc  --> *likely useful*  
  resources.py   --> *compiled version, likely useful*  
  form.ui        --> *likely useful*  
  form.py        --> *compiled version, likely useful*
```

Care este semnificaia fiierelor:

- `__init__.py` = Punctul de plecare al plugin-ului. Acesta trebuie să aibă metoda `classFactory()` și poate avea orice alt cod de inițializare.
- `mainPlugin.py` = Principalul codul lucrativ al plugin-ului. Conține toate informațiile cu privire la acțiunile plugin-ului și ale codului principal.
- `resources.qrc` = Documentul .xml creat de Qt Designer. Conține căi relative la resursele formelor.
- `resources.py` = Traducerea fiierului .qrc descris mai sus.
- `form.ui` = GUI-ul creat de Qt Designer.
- `form.py` = Traducerea form.ui descris mai sus.
- `metadata.txt` = Required for QGIS >= 1.8.0. Conține general info, version, name and some other metadata used by plugins website and plugin infrastructure. Since QGIS 2.0 the metadata from `__init__.py` are not accepted anymore and the `metadata.txt` is required.

Aici este o modalitate on-line, automată, de creare a fiierelor de bază (carcase) pentru un plugin tipic QGIS Python.

De asemenea, există un plugin QGIS numit [Plugin Builder](#) care creează un ablon de plugin QGIS și nu are nevoie de conexiune la internet. Aceasta este opțiunea recomandată, atât timp cât produce surse compatibile 2.0.

Warning: Dacă aveți de gând să încărcați plugin-ul la *Depozitul oficial al plugin-urilor python* trebuie să verificați că plugin-ul urmează anumite reguli suplimentare, necesare pentru *Validare* plugin

13.2 Conținutul Plugin-ului

Aici puteți găsi informații și exemple despre ceea ce să adăugați în fiecare dintre fiierle din structura de fiere descrisă mai sus.

13.2.1 | Metadatele plugin-ului

În primul rând, managerul de plugin-uri are nevoie de preluarea câtorva informații de bază despre plugin, cum ar fi numele, descrierea etc. Fiierul `metadata.txt` este locul potrivit pentru a reține această informație.

Important: Toate metadatele trebuie să fie în codificarea UTF-8.

Numele metadatei	Obligato-riu	Note
nume	True	un ir scurt coninând numele pluginului
qgisMini- mumVersion	True	notaie cu punct a versiunii minime QGIS
qgisMaxi- mumVersion	False	notaie cu punct a versiunii maxime QGIS
descriere	True	scurt text care descrie plugin-ul, HTML nefiind permis
despre	False	text mai lung care descrie plugin-ul în detalii, HTML nefiind permis
versiune	True	scurt ir cu versiunea notată cu punct
autor	True	nume autor
email	True	email of the author, will <i>not</i> be shown on the web site
jurnalul schimbărilor experimental	False	ir, poate fi pe mai multe linii, HTML nefiind permis
învechit	False	semnalizator boolean, <i>True</i> sau <i>False</i>
etichete	False	semnalizator boolean, <i>True</i> sau <i>False</i> , se aplică întregului plugin i nu doar la versiunea încărcată
pagina de casă	False	o listă de valori separate prin virgulă, spaiele fiind permise în interiorul etichetelor individuale
depozit	False	o adresă URL validă indicând spre pagina plugin-ului dvs.
tracker	False	o adresă URL validă pentru depozitul de cod sursă
pictogramă	False	o adresă validă pentru bilete i rapoartare de erori
categorie	False	a file name or a relative path (relative to the base folder of the plugin's compressed package)
		una din valorile <i>Raster</i> , <i>Vector</i> , <i>Bază de date</i> i <i>Web</i>

În mod implicit, plugin-urile sunt plasate în meniul *Plugin-uri* (vom vedea în seciunea următoare cum se poate adăuga o intrare de meniu pentru plugin-ul dvs.), dar ele pot fi, de asemenea, plasate i în meniurile *Raster*, *Vector*, *Database* i *Web*.

Există o “categorie” de intrare de metadate corespunzătoare pentru a preciza că, astfel, plugin-ul poate fi clasificat în consecină. Această intrare este folosită ca indiciu pentru utilizatori i le spune unde (în care meniu), poate fi găsit plugin-ul. Valorile permise pentru “categorie” sunt: *Vector*, *Raster*, *Baza de date* sau *Web*. De exemplu, dacă plugin-ul va fi disponibil din meniul *Raster*, adăugai următoarele în `metadata.txt`

```
category=Raster
```

Note: În cazul în care valoarea `qgisMaximumVersion` este vidă, ea va fi setată automat la versiunea majoră plus `0.99` încărcată în depozitul *Depozitul oficial al plugin-urilor python*.

Un exemplu pentru acest `metadata.txt`

```
; the next section is mandatory

[general]
name>HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
; end of mandatory metadata

; start of optional metadata
category=Raster
```

```
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

; Tags are in comma separated value format, spaces are allowed within the
; tag name.
; Tags should be in English language. Please also check for existing tags and
; synonyms before creating a new one.
tags=wkt,raster,hello world

; these metadata can be empty, they will eventually become mandatory.
homepage=http://www.itopen.it
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
icon=icon.png

; experimental flag (applies to the single version)
experimental=True

; deprecated flag (applies to the whole plugin and not only to the uploaded version)
deprecated=False

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=2.0
```

13.2.2 `__init__.py`

Acest fiier este necesar pentru sistemul de import al Python. De asemenea, QGIS necesită ca acest fiier să conțină o funcție `classFactory()`, care este apelată atunci când plugin-ul se încarcă în QGIS. Acesta primește referința la o instanță a `QgisInterface` și trebuie să returneze instanța clasei plugin-ului din `mainplugin.py` — în cazul nostru numindu-se `TestPlugin` (a se vedea mai jos). Acesta este modul în care `__init__.py` ar trebui să arate

```
def classFactory(iface):
    from mainPlugin import TestPlugin
    return TestPlugin(iface)
```

```
## any other initialisation needed
```

13.2.3 `mainPlugin.py`

Aici este locul în care se întâmplă magia, iată rezultatul acesteia: (de exemplu `mainPlugin.py`)

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

    def __init__(self, iface):
        # save reference to the QGIS interface
        self.iface = iface

    def initGui(self):
        # create action that will start plugin configuration
```



```

self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self iface.mainWindow)
self.action.setObjectName("testAction")
self.action.setWhatsThis("Configuration for test plugin")
self.action.setStatusTip("This is status tip")
QObject.connect(self.action, SIGNAL("triggered()"), self.run)

# add toolbar button and menu item
self iface.addToolBarIcon(self.action)
self iface.addPluginToMenu("&Test plugins", self.action)

# connect to signal renderComplete which is emitted when canvas
# rendering is done
QObject.connect(self iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def unload(self):
    # remove the plugin menu item and icon
    self iface.removePluginMenu("&Test plugins", self.action)
    self iface.removeToolBarIcon(self.action)

    # disconnect from signal of the canvas
    QObject.disconnect(self iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def run(self):
    # create and show a configuration dialog or something similar
    print "TestPlugin: run called!"

def renderTest(self, painter):
    # use painter for drawing to map canvas
    print "TestPlugin: renderTest called!"

```

Singurele funcții care trebuie să existe în fișierul sursă al plugin-ului principal (de exemplu mainPlugin.py) sunt:

- `__init__` -> care oferă acces la interfața QGIS
- `initGui()` -> apelat atunci când plugin-ul este încărcat
- `unload()` -> apelat atunci când plugin-ul este descărcat

Puteți vedea că în exemplul de mai sus se folosește `addPluginToMenu()`. Aceasta va adăuga aciunea meniului corespunzător la meniul *Plugins*. Există metode alternative pentru a adăuga aciunea la un alt meniu. Iată o listă a acestor metode:

- `addPluginToRasterMenu()`
- `addPluginToVectorMenu()`
- `addPluginToDatabaseMenu()`
- `addPluginToWebMenu()`

Toate acestea au aceeași sintaxă ca metoda `addPluginToMenu()`.

Adăugarea unui meniu la plugin-ul dvs. printr-una din metodele predefinite este recomandată pentru a păstra coerența în stilul de organizare a plugin-urilor. Cu toate acestea, puteți adăuga grupul dvs. de meniuri personalizate direct în bara de meniu, așa cum demonstrează următorul exemplu :

```

def initGui(self):
    self.menu = QMenu(self iface.mainWindow())
    self.menu.setObjectName("testMenu")
    self.menu.setTitle("MyMenu")

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self iface.mainWindow)
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")

```

```
QObject.connect(self.action, SIGNAL("triggered()"), self.run)
self.menu.addAction(self.action)

menuBar = self iface.mainWindow().menuBar()
menuBar.insertMenu(self iface.firstRightStandardMenu().menuAction(), self.menu)

def unload(self):
    self.menu.deleteLater()
```

Nu uitai să stabilești pentru `QObject` și `QMenu` `objectName` un nume specific plugin-ului dvs, astfel încât acesta să poată fi personalizat.

13.2.4 Fier de resurse

Puteți vedea că în `initGui()` am folosit o pictogramă din fiierul de resurse (denumit `resources.qrc`, în cazul nostru)

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

It is good to use a prefix that will not collide with other plugins or any parts of QGIS, otherwise you might get resources you did not want. Now you just need to generate a Python file that will contain the resources. It's done with **pyrcc4** command

```
pyrcc4 -o resources.py resources.qrc
```

i asta e tot ... nimic complicat :)

Dacă ai făcut totul corect, ar trebui să găsești și să încarci plugin-ul în managerul de plugin-uri și să vezi un mesaj în consolă, atunci când este selectat meniul adecvat sau pictograma din bara de instrumente.

Când lucrezi la un plug-in real, este înțelept să stoci plugin-ul într-un alt director (de lucru), și să creai un fiier `make` care va genera UI + fierele de resurse și să instalezi plugin-ul în instalarea QGIS.

13.3 Documentație

Documentația pentru plugin poate fi scrisă ca fiere HTML. Modulul `qgis.utils` oferă o funcție, `showPluginHelp()`, care se va deschide navigatorul de fiere, în același mod ca și altă fereastră de ajutor QGIS.

The `showPluginHelp()` function looks for help files in the same directory as the calling module. It will look for, in turn, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` and `index.html`, displaying whichever it finds first. Here `ll_cc` is the QGIS locale. This allows multiple translations of the documentation to be included with the plugin.

Funcția `showPluginHelp()` poate lua, de asemenea, parametrii `packageName`, care identifică plugin-ul specific pentru care va fi afișat ajutorul, numele de fiere, care poate înlocui 'index' în numele fiierelor în care se caută, și seciunea, care este numele unei ancore HTML în documentul în care se va poziționa browser-ul.

Setările IDE pentru scrierea i depanarea de plugin-uri

Dei fiecare programator preferă un anumit editor IDE/Text, iată câteva recomandări de setare a unor IDE-uri populare, pentru scrierea i depanarea plugin-urilor Python specifice QGIS.

14.1 O notă privind configurarea IDE-ului în Windows

On Linux there is no additional configuration needed to develop plug-ins. But on Windows you need to make sure you that you have the same environment settings and use the same libraries and interpreter as QGIS. The fastest way to do this, is to modify the startup batch file of QGIS.

If you used the OSGeo4W Installer, you can find this under the bin folder of your OSGeoW install. Look for something like `C:\OSGeo4W\bin\qgis-unstable.bat`.

Pentru utilizarea IDE-ului Pyscripter, iată ce avei de făcut:

- Make a copy of `qgis-unstable.bat` and rename it `pyscripter.bat`.
- Deschidei-o într-un editor. Apoi elimini ultima linie, cea care startează QGIS.
- Add a line that points to the your Pyscripter executable and add the commandline argument that sets the version of Python to be used (2.7 in the case of QGIS 2.0)
- De asemenea, adăugai i un argument care să indice folderul unde poate găsi Pyscripter dll-ul Python folosit de către QGIS, acesta aflându-se în folderul bin al instalării OSGeoW

```
@echo off
SET OSGEO4W_ROOT=C:\OSGeo4W
call "%OSGEO4W_ROOT%\bin\o4w_env.bat
call "%OSGEO4W_ROOT%\bin\gdal16.bat
@echo off
path %PATH%;%GISBASE%\bin
Start C:\pyscripter\pyscripter.exe --python25 --pythondllpath=C:\OSGeo4W\bin
```

Acum, când vei efectua dublu clic pe acest fiier batch, el va starta Pyscripter, având calea corectă.

More popular than Pyscripter, Eclipse is a common choice among developers. In the following sections, we will be explaining how to configure it for developing and testing plugins. To prepare your environment for using Eclipse in Windows, you should also create a batch file and use it to start Eclipse.

To create that batch file, follow these steps.

- Locate the folder where file: `qgis_core.dll` resides in. Normally this is `C:\OSGeo4W\apps\qgis\bin`, but if you compiled your own QGIS application this is in your build folder in `output/bin/RelWithDebInfo`
- Localizai executabilul `eclipse.exe`.
- Creai următorul script i folosești-l pentru a starta Eclipse, atunci când dezvoltai plugin-uri QGIS.

```
call "C:\OSGeo4W\bin\o4w_env.bat"  
set PATH=%PATH%;C:\path\to\your\qgis_core.dll\parent\folder  
C:\path\to\your\eclipse.exe
```

14.2 Depanare cu ajutorul Eclipse i PyDev

14.2.1 Instalare

Pentru a utiliza Eclipse, asigurai-vă că ai instalat următoarele

- Eclipse
- Aptana Eclipse Plugin or PyDev
- QGIS 2.0

14.2.2 Pregătirea QGIS

There is some preparation to be done on QGIS itself. Two plugins are of interest: *Remote Debug* and *Plugin reloader*.

- Go to *Plugins* → *Fetch python plugins*
- Search for *Remote Debug* (at the moment it's still experimental, so enable experimental plugins under the Options tab in case it does not show up). Install it.
- De asemenea, căutați *Plugin Reloader* i instalai-l. Acest lucru vă va permite să reîncărcați un plug-in, fără a fi necesare închiderea i repornirea QGIS.

14.2.3 Configurarea Eclipse

În Eclipse, creai un nou proiect. Putei să selectai *General Project* i să legai ulterior sursele dvs. reale, aa că nu prea contează unde plasai acest proiect.

Now right click your new project and choose *New* → *Folder*.

Click [**Advanced**] and choose *Link to alternate location (Linked Folder)*. In case you already have sources you want to debug, choose these, in case you don't, create a folder as it was already explained

Acum, în fereastra *Project Explorer*, va apărea arborele sursă i vei putea începe să lucrezi la cod. Aveți disponibile deja evidențierea sintaxei i toate celelalte instrumente puternice din IDE.

14.2.4 Configurarea depanatorului

Pentru a vedea depanatorul la lucru, comutați în perspectiva Depanare din Eclipse (*Window* → *Open Perspective* → *Other* → *Debug*).

Acum, pornii serverul de depanare PyDev, alegând *PyDev* → *Start Debug Server*.

Eclipse is now waiting for a connection from QGIS to its debug server and when QGIS connects to the debug server it will allow it to control the python scripts. That's exactly what we installed the *Remote Debug* plugin for. So start QGIS in case you did not already and click the bug symbol .

Now you can set a breakpoint and as soon as the code hits it, execution will stop and you can inspect the current state of your plugin. (The breakpoint is the green dot in the image below, set one by double clicking in the white space left to the line you want the breakpoint to be set)

Un aspect foarte interesant este faptul că puteți utiliza consola de depanare. Asigurați-vă că execuția este, în mod curent, stopată, înainte de a continua.

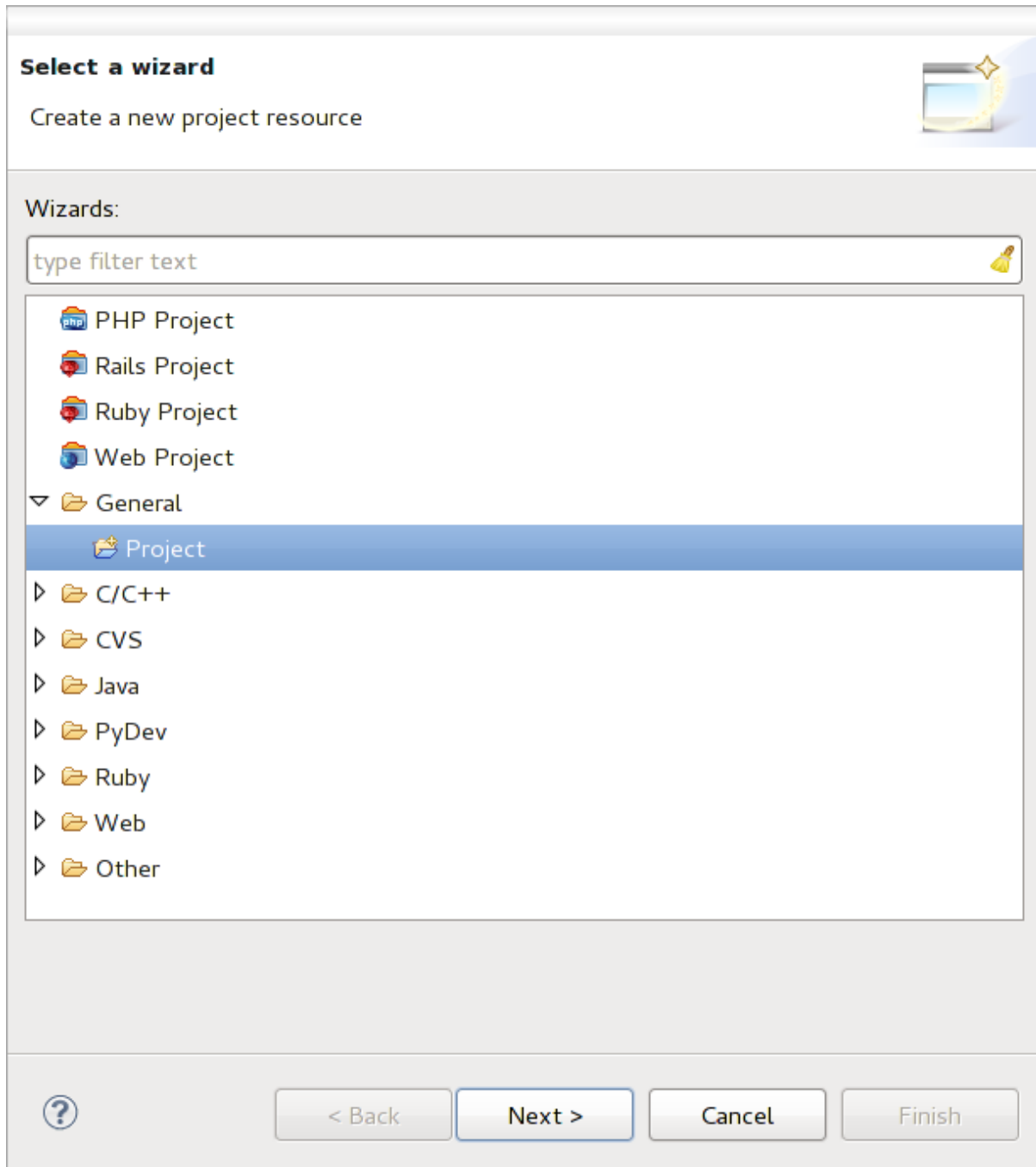


Figure 14.1: Proiectul Eclipse

```

87         self.verticalExaggerationChanged.emit(val)
88
89     def printProfile(self):
90         printer = QPrinter( QPrinter.HighResolution )
91         printer.setOutputFormat( QPrinter.PdfFormat )
92         printer.setPaperSize( QPrinter.A4 )
93         printer.setOrientation( QPrinter.Landscape )
94
95         printPreviewDlg = QPrintPreviewDialog( )
96         printPreviewDlg.paintRequested.connect( self.printRequested )
97
98         printPreviewDlg.exec_()
99
100    @pyqtSlot( QPrinter )
101    def printRequested( self, printer ):
102        self.webView.print_( printer )
103

```

Figure 14.2: Punct de întrerupere

Deschideți fereastra consolei (*Window* → *Show view*). Se va afișa consola *Debug Server*, ceea ce nu este prea interesant. În schimb, butonul [**Open Console**] permite trecerea la mult mai interesanta consolă de depanare PyDev. Faceți clic pe săgeata de lângă butonul [**Open Console**] și alegeți *PyDev Console*. Se deschide o fereastră care vă va întreba ce consolă doriți să deschideți. Alegeți *PyDev Debug Console*. În cazul când aceasta este gri, vă cere să porniți depanatorul și să selectați un cadru valid, asigurați-vă că ai atașat depanatorul la distanță, iar în prezent sunteți pe un punct de întrerupere.

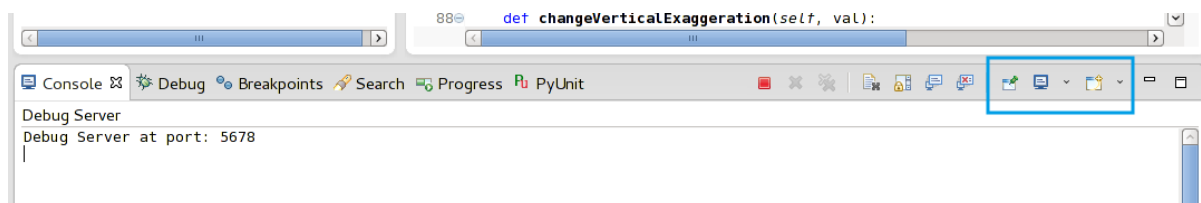


Figure 14.3: Consola de depanare PyDev

Acum aveți o consolă interactivă care vă permite să testați orice comenzi din interior, în contextul actual. Puteți manipula variabile, să efectuați apeluri API sau orice altceva.

Un pic enervant este faptul că, de fiecare dată când introduceți o comandă, consola comută înapoi la serverul de depanare. Pentru a opri acest comportament, aveți posibilitatea să faceți clic pe butonul *Pin Console* din pagina serverului de depanare, pentru reinerea acestei decizii, cel puțin pentru sesiunea de depanare curentă.

14.2.5 Configurați Eclipse pentru a înțelege API-ul

O caracteristică facilă este de a pregăti Eclipse pentru API-ul QGIS. Aceasta vă permite verificarea eventualelor greeli de ortografie din cadrul codului. Dar nu doar atât, vă permite ca Eclipse să autocompleteze din importurile către apelurile API.

Pentru a face acest lucru, Eclipse analizează fierele bibliotecii QGIS și primește toate informațiile de acolo. Singurul lucru pe care trebuie să-l faceți este de a-i indica lui Eclipse unde să găsească bibliotecile.

Faceți clic pe *Window* → *Preferences* → *PyDev* → *Interpreter* → *Python*.

Vei vedea interpretorul de python (pe moment versiunea 2.7) configurat, în partea de sus a ferestrei și unele fișiere în partea de jos. Fișierele interesante pentru noi sunt *Libraries* și *Forced Builtins*.

În primul rând deschideți fila *Libraries*. Adăugați un folder nou și selectați folderul python al aplicației QGIS instalate. Dacă nu știți unde se află acest director (acesta nu este folderul plugin-urilor) deschideți QGIS, startați o consolă python și pur și simplu introduceți `qgis`, apoi apăsați `Enter`. Acest lucru vă va arăta care modul QGIS este folosit, precum și calea sa. tergeți `/qgis/__init__.pyc` și veți obține calea pe care o căutați.

You should also add your plugins folder here (on Linux it is `~/ .qgis/python/plugins`).

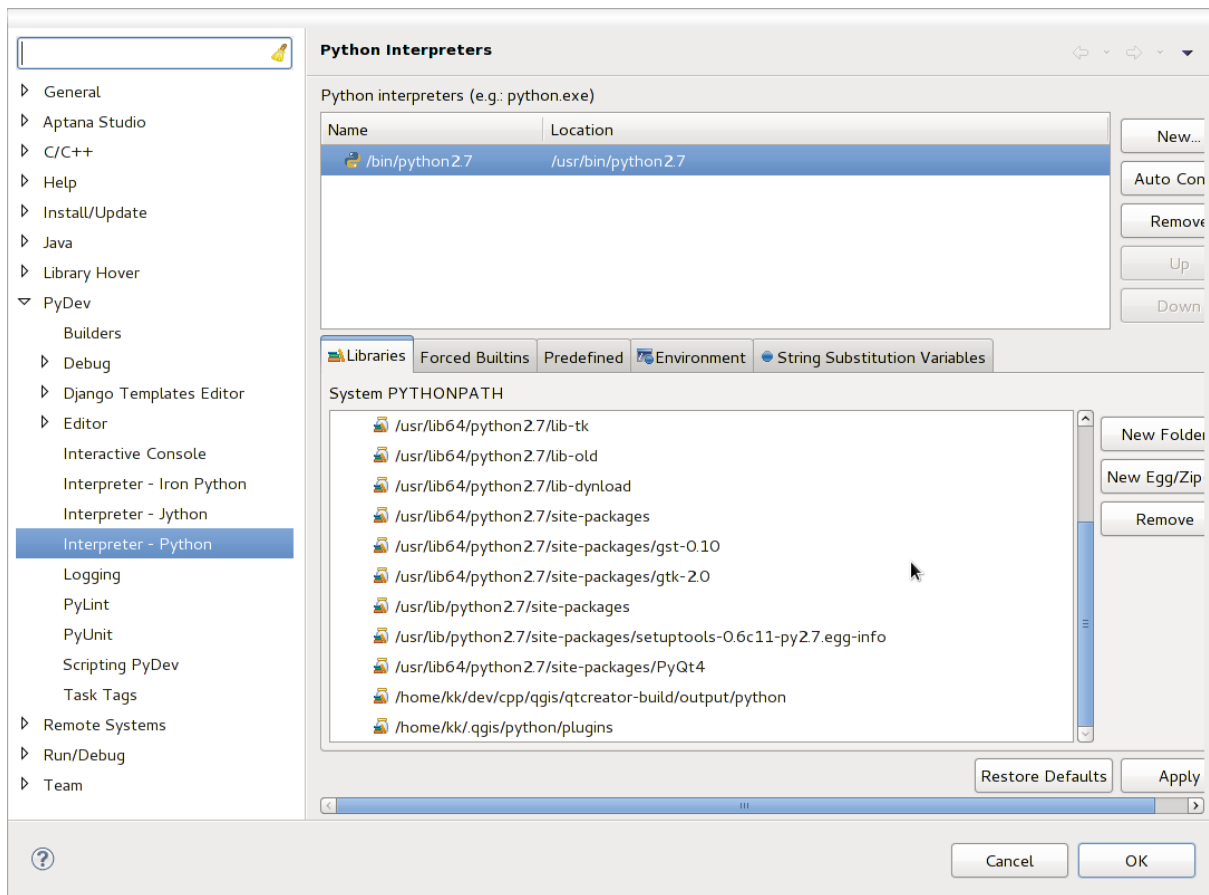


Figure 14.4: Consola de depanare PyDev

Next jump to the *Forced Builtins* tab, click on *New...* and enter `qgis`. This will make Eclipse parse the QGIS API. You probably also want eclipse to know about the PyQt4 API. Therefore also add PyQt4 as forced builtin. That should probably already be present in your libraries tab.

Facei clic pe *OK* i ai terminat.

Note: every time the QGIS API changes (e.g. if you're compiling QGIS master and the SIP file changed), you should go back to this page and simply click *Apply*. This will let Eclipse parse all the libraries again.

Pentru o altă setare posibilă de Eclipse, pentru a lucra cu plugin-urile Python QGIS, verificai [acest link](#)

14.3 Depanarea cu ajutorul PDB

Dacă nu folosii un IDE, cum ar fi Eclipse, putei depana folosind PDB, urmând aceti pai.

Mai întâi, adăugai acest cod în locul în care dorii depanarea

```
# Use pdb for debugging
import pdb
# These lines allow you to set a breakpoint in the app
pyqtRemoveInputHook()
pdb.set_trace()
```

Apoi executai QGIS din linia de comandă.

On Linux do:

```
$ ./Qgis
```

On Mac OS X do:

```
$/Applications/Qgis.app/Contents/MacOS/Qgis
```

Iar când aplicaia atinge punctul de întrerupere avei posibilitatea de a tasta în consolă!

DE EFECTUAT: Adăugai informații pentru testare

Utilizarea straturilor plugin-ului

Dacă plugin-ul dvs. folosește propriile metode de a randa un strat de hartă, scrierea propriului tip de strat, bazat pe `QgsPluginLayer`, ar putea fi cea mai indicată.

DE EFECTUAT: Verificai corectitudinea și elaborai cazuri de utilizare corectă pentru `QgsPluginLayer`, ...

15.1 Subclasarea `QgsPluginLayer`

Mai jos este un exemplu minimal de implementare pentru `QgsPluginLayer`. Acesta este un extras din [Exemplu de plugin filigran](#)

```
class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

    def __init__(self):
        QgsPluginLayer.__init__(self, WatermarkPluginLayer.LAYER_TYPE, "Watermark plugin layer")
        self.setValid(True)

    def draw(self, rendererContext):
        image = QImage("myimage.png")
        painter = rendererContext.painter()
        painter.save()
        painter.drawImage(10, 10, image)
        painter.restore()
        return True
```

De asemenea, pot fi adăugate metode pentru citirea și scrierea de informații specifice, în fișierul proiectului

```
def readXml(self, node):
    pass

def writeXml(self, node, doc):
    pass
```

Când încarci un proiect care conține un astfel de strat, este nevoie de o fabrică de clase

```
class WatermarkPluginLayerType(QgsPluginLayerType):

    def __init__(self):
        QgsPluginLayerType.__init__(self, WatermarkPluginLayer.LAYER_TYPE)

    def createLayer(self):
        return WatermarkPluginLayer()
```

Puteți adăuga, de asemenea, codul pentru afișarea de informații personalizate în proprietățile stratului

```
def showLayerProperties(self, layer):  
    pass
```

Compatibilitatea cu versiunile QGIS anterioare

16.1 Meniul plugin-ului

Dacă plasai intrările de meniu ale plugin-ului într-unul dintre noile meniuri (*Raster*, *Vector*, *Database* sau *Web*), va trebui să modificai codul funcțiilor `initGui()` și `unload()`. Din moment ce aceste noi meniuri sunt disponibile începând de la QGIS 2.0, primul pas este de a verifica faptul că versiunea de QGIS are toate funcțiile necesare. Dacă noile meniuri sunt disponibile, vom plasa pluginul nostru în cadrul acestui meniu, altfel vom folosi vechiul meniu *Plugins*. Iată un exemplu pentru meniul *Raster*

```
def initGui(self):
    # create action that will start plugin configuration
    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWindow)
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

    # check if Raster menu available
    if hasattr(self.iface, "addPluginToRasterMenu"):
        # Raster menu and toolbar available
        self.iface.addRasterToolBarIcon(self.action)
        self.iface.addPluginToRasterMenu("&Test plugins", self.action)
    else:
        # there is no Raster menu, place plugin under Plugins menu as usual
        self.iface.addToolBarIcon(self.action)
        self.iface.addPluginToMenu("&Test plugins", self.action)

    # connect to signal renderComplete which is emitted when canvas rendering is done
    QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def unload(self):
    # check if Raster menu available and remove our buttons from appropriate
    # menu and toolbar
    if hasattr(self.iface, "addPluginToRasterMenu"):
        self.iface.removePluginRasterMenu("&Test plugins", self.action)
        self.iface.removeRasterToolBarIcon(self.action)
    else:
        self.iface.removePluginMenu("&Test plugins", self.action)
        self.iface.removeToolBarIcon(self.action)

    # disconnect from signal of the canvas
    QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)
```

Lansarea plugin-ului dvs.

O dată ce plugin-ul este gata și credei că el ar putea fi de ajutor pentru unii utilizatori, nu ezitai să-l încarci la *Depozitul oficial al plugin-urilor python*. Pe acea pagină poți găsi instrucțiuni de împachetare și de pregătire a plugin-ului, pentru a lucra bine cu programul de instalare. Sau, în cazul în care ai dori să înființezi un depozit propriu pentru plugin-uri, creai un simplu fișier XML, care va lista plugin-urile și metadatele lor, exemplu pe care îl poți vedea în *depozite pentru plugin-uri*.

Vă rugăm să acordați o grijă deosebită următoarelor recomandări:

17.1 Metadate și nume

- evitați folosirea unui nume prea asemănător cu cel al plugin-urilor existente
- dacă plugin-ul are o funcționalitate similară cu cea a unui plugin existent, vă rugăm să explicați diferențele în câmpul *Despre*, astfel încât utilizatorul va ști pe care să-l folosească, fără a fi nevoie de instalare și testare
- evitați repetarea cuvântului “plugin”, în denumirea unui plugin
- utilizați câmpul *descriere* din metadate pentru o descriere de 1 linie, și câmpul *Despre* pentru instrucțiuni mai detaliate
- includeți un depozit de cod, un monitor de erori, și o pagină de start; astfel, va spori considerabil posibilitatea de colaborare, aceasta făcându-se foarte ușor cu ajutorul infrastructurilor web disponibile (GitHub, GitLab, BitBucket, etc.)
- alegeți etichetele cu grijă: evitați-le pe cele neinformative (ex: *vector*), preferându-le pe cele deja folosite de către alții (a se vedea site-ul plugin-urilor)
- adăugați o pictogramă adecvată, și nu o lăsați pe cea implicită; vedeți interfața QGIS pentru o sugestie despre stilul de utilizat

17.2 Codul și ajutorul

- nu includeți fișierul generat (*ui_*.py*, *resources_rc.py*, fișiere de ajutor generate...) și chestii inutile (ex: *.gitignore*) în depozit
- adăugați pluginul în meniul corespunzător (*Vector*, *Raster*, *Web*, *Bază de date*)
- atunci când este cazul (plugin-uri efectuând analize), luați în considerare adăugarea plugin-ului ca subplugin al cadrului de *Procesare*: acest lucru va permite utilizatorilor să-l rulați în lot, să-l integrați în fluxurile de lucru mai complexe, eliberându-vă de povara proiectării unei interfețe
- includeți cel puțin documentația minimă și, dacă este util pentru testare și înțelegere, datele eantion.

17.3 Depozitul oficial al plugin-urilor python

Puteți găsi depozitul *oficial* al plugin-urilor python la <http://plugins.qgis.org/>.

Pentru a folosi depozitul oficial, trebuie să obțineți un ID OSGEO din portalul web OSGEO.

O dată ce ai încărcat plugin-ul, acesta va fi aprobat de către un membru al personalului și vei primi o notificare.

DE EFECTUAT: Introduceți un link către documentul guvernanei

17.3.1 Permisuni

Aceste reguli au fost implementate în depozitul oficial al plugin-urilor:

- fiecare utilizator înregistrat poate adăuga un nou plugin
- membrii *staff-ului* pot aproba sau dezaproba toate versiunile plugin-ului
- utilizatorii care au permisiunea specială *plugins.can_approve* au versiunile pe care le încarcă aprobate în mod automat
- utilizatorii care au permisiunea specială *plugins.can_approve* pot aproba versiunile încărcate de către alții, atât timp cât aceștia sunt prezenți în lista *proprietarilor* de plugin-uri
- un anumit plug-in pot fi terși și editat doar de utilizatorii *staff-ului* și de către *proprietarii* plugin-ului
- în cazul în care un utilizator fără permisiunea *plugins.can_approve* încarcă o nouă versiune, versiunea plugin-ului nu va fi aprobată, din start.

17.3.2 Managementul încrederii

Membrii personalului pot acorda *încredere* creatorilor de plugin-uri, bifând permisiunea *plugins.can_approve* în cadrul front-end-ului.

Detaliile despre plugin oferă legături directe pentru a crește încrederea în creatorul sau *proprietarul* plugin-ului.

17.3.3 Validare

Metadatele plugin-ului sunt importate automat din pachetul arhivat și sunt validate, la încărcarea plugin-ului.

Iată câteva reguli de validare pe care ar trebui să le cunoașteți atunci când doriți să încărcați un plugin în depozitul oficial:

1. numele folderului principal, care include plugin-ul, trebuie să conțină numai caracterele ASCII (A-Z și a-z), cifre, caractere de subliniere (`_`), minus (`-`) și, de asemenea, nu poate începe cu o cifră
2. `metadata.txt` este necesar
3. toate metadatele necesare, menționate în *tabela de metadate* trebuie să fie prezente
4. the *version* metadata field must be unique

17.3.4 Structura plugin-ului

Conform regulilor de validare, pachetul comprimat (.zip) al plugin-ului trebuie să aibă o structură specifică, pentru a fi validat ca plugin funcțional. Deoarece plugin-ul va fi dezarhivat în interiorul directorului de plugin-uri ale utilizatorului, el trebuie să aibă propriul director în interiorul fișierului zip, pentru a nu interfera cu alte plugin-uri. Fișierele obligatorii sunt: `metadata.txt` și `__init__.py`. Totuși, ar fi frumos să existe un README și, desigur, o pictogramă care să reprezinte pluginul (`resources.qrc`). Iată un exemplu despre modul în care ar trebui să arate un plugin.zip.

```
plugin.zip
  pluginfolder/
    |-- i18n
    |   |-- translation_file_de.ts
    |-- img
    |   |-- icon.png
    |   |-- iconsource.svg
    |-- __init__.py
    |-- Makefile
    |-- metadata.txt
    |-- more_code.py
    |-- main_code.py
    |-- README
    |-- resources.qrc
    |-- resources_rc.py
    |-- ui_Qt_user_interface_file.ui
```

Fragmente de cod

Această seciune conține fragmente de cod, menite să faciliteze dezvoltarea plugin-urilor.

18.1 Cum să apelăm o metodă printr-o combinație rapidă de taste

În plug-in adăugai în `initGui()`

```
self.keyAction = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.keyAction, "F7") # action1 triggered by F7 key
self.iface.addPluginToMenu("&Test plugins", self.keyAction)
QObject.connect(self.keyAction, SIGNAL("triggered()"), self.keyActionF7)
```

În `unload()` adă

```
self.iface.unregisterMainWindowAction(self.keyAction)
```

Metoda care este chemată atunci când se apasă tasta F7

```
def keyActionF7(self):
    QMessageBox.information(self.iface.mainWindow(), "Ok", "You pressed F7")
```

18.2 Inversarea Stării Straturilor

Începând de la QGIS 2.4, un nou API permite accesul direct la arborele straturilor din legendă. Exemplul următor prezintă modul în care se poate inversa vizibilitatea stratului activ

```
root = QgsProject.instance().layerTreeRoot()
node = root.findLayer(iface.activeLayer().id())
new_state = Qt.Checked if node.isVisible() == Qt.Unchecked else Qt.Unchecked
node.setVisible(new_state)
```

18.3 Cum să accesezi tabelul de atribute al entităților selectate

```
def changeValue(self, value):
    layer = self.iface.activeLayer()
    if layer:
        nF = layer.selectedFeatureCount()
        if (nF > 0):
            layer.startEditing()
            ob = layer.selectedFeaturesIds()
            b = QVariant(value)
            if (nF > 1):
```

```
    for i in ob:
        layer.changeAttributeValue(int(i), 1, b) # 1 being the second column
    else:
        layer.changeAttributeValue(int(ob[0]), 1, b) # 1 being the second column
    layer.commitChanges()
    else:
        QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select at least one feature")
    else:
        QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select a layer")
```

Metoda necesită un parametru (noua valoare pentru câmpul atribut al entităi(lor) selectate) i poate fi apelat de către

`self.changeValue(50)`

Biblioteca de analiză a reelelor

Începând cu revizia [ee19294562](#) (QGIS \geq 1.8) noua bibliotecă de analiză de reea a fost adăugată la biblioteca de analize de bază din QGIS. Biblioteca:

- creează graful matematic din datele geografice (straturi vectoriale de tip polilinie)
- implementează metode de bază din teoria grafurilor (în prezent, doar algoritmul lui Dijkstra)

Biblioteca analizelor de reea a fost creată prin exportarea funcțiilor de bază ale plugin-ului RoadGraph, iar acum aveți posibilitatea să-i utilizați metodele în plugin-uri sau direct în consola Python.

19.1 Informații generale

Pe scurt, un caz tipic de utilizare poate fi descris astfel:

1. crearea grafului din geodate (de obicei un strat vectorial de tip polilinie)
2. rularea analizei grafului
3. folosirea rezultatelor analizei (de exemplu, vizualizarea lor)

19.2 Construirea unui graf

Primul lucru pe care trebuie să-l faceți — este de a pregăti datele de intrare, ceea ce înseamnă conversia stratului vectorial într-un graf. Toate acțiunile viitoare vor folosi acest graf, și nu stratul.

Ca și sursă putem folosi orice strat vectorial de tip polilinie. Nodurile poliliniilor devin noduri ale grafului, segmentele poliliniilor reprezentând marginile grafului. În cazul în care mai multe noduri au aceleași coordonate, atunci ele sunt în același nod al grafului. Astfel, două linii care au un nod comun devin conectate între ele.

În plus, în timpul creării grafului este posibilă “fixarea” (“legarea”) de stratul vectorial de intrare a oricărui număr de puncte suplimentare. Pentru fiecare punct suplimentar va fi găsită o potrivire — cel mai apropiat nod sau cea mai apropiată muchie a grafului. În ultimul caz muchia va fi divizată iar noul nod va fi adăugat.

Atributele stratului vectorial și lungimea unei muchii pot fi folosite ca proprietăți ale marginii.

Convertorul din strat vectorial în graf este dezvoltat folosind modelul de programare al **Constructorului**. De construirea grafului răspunde aa-numitul Director. Există doar un singur Director pentru moment: `QgsLineVectorLayerDirector`. Directorul stabilește setările de bază ale Constructorului, care vor fi folosite pentru a construi un graf dintr-un strat vectorial de tip linie. În prezent, ca și în cazul Directorului, există doar un singur Constructor: `QgsGraphBuilder`, care creează obiecte <http://qgis.org/api/classQgsGraph.html> ‘_’. Este posibil să doriți implementarea propriilor constructori care să construiască grafuri compatibile cu bibliotecile, cum ar fi BGL sau NetworkX.

Pentru a calcula proprietăile marginii este utilizată **strategia modelului de programare**. Pentru moment doar strategia `QgsDistanceArcProperter` este disponibilă, care ia în considerare lungimea traseului. Putei implementa propria strategie, care va folosi toi parametrii necesari. De exemplu, plugin-ul `RoadGraph` folosește strategia care calculează timpul de călătorie, folosind lungimea muchiei și valoarea vitezei din atribute.

Este timpul de a aprofunda acest proces.

Înainte de toate, pentru a utiliza această bibliotecă ar trebui să importăm modulul `networkanalysis`

```
from qgis.networkanalysis import *
```

Apoi, câteva exemple pentru crearea unui director

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsLineVectorLayerDirector(vLayer, -1, '', '', '', 3)

# use field with index 5 as source of information about road direction.
# one-way roads with direct direction have attribute value "yes",
# one-way roads with reverse direction have the value "1", and accordingly
# bidirectional roads have "no". By default roads are treated as two-way.
# This scheme can be used with OpenStreetMap data
director = QgsLineVectorLayerDirector(vLayer, 5, 'yes', '1', 'no', 3)
```

Pentru a construi un director ar trebui să transmitem stratul vectorial, care va fi folosit ca sursă pentru structura grafului și informațiile despre micările permise pe fiecare segment de drum (circulație unilaterală sau bilaterală, sens direct sau invers). Acest apel arată în felul următor

```
director = QgsLineVectorLayerDirector(vl, directionFieldId,
                                     directDirectionValue,
                                     reverseDirectionValue,
                                     bothDirectionValue,
                                     defaultDirection)
```

Iată lista completă a ceea ce înseamnă acești parametri:

- `vl` — stratul vectorial utilizat pentru a construi graf
- `directionFieldId` — indexul câmpului din tabelul de atribute, în care sunt stocate informații despre direcțiile drumurilor. Dacă este `-1`, atunci aceste informații nu se folosesc deloc. Număr întreg.
- `directDirectionValue` — valoarea câmpului pentru drumurile cu sens direct (trecere de la primul punct de linie la ultimul). ir de caractere.
- `reverseDirectionValue` — valoarea câmpului pentru drumurile cu sens invers (în mișcare de la ultimul punct al liniei până la primul). ir de caractere.
- `bothDirectionValue` — valoarea câmpului pentru drumurile bilaterale (pentru astfel de drumuri putem trece de la primul la ultimul punct și de la ultimul la primul). ir de caractere.
- `defaultDirection` — direcția implicită a drumului. Această valoare va fi folosită pentru acele drumuri în care câmpul `directionFieldId` nu este setat sau are o valoare diferită de oricare din cele trei valori specificate mai sus. Număr întreg. 1 indică sensul direct, 2 indică sensul invers, iar 3 indică ambele sensuri.

Este necesară, apoi, crearea unei strategii pentru calcularea proprietăților marginii

```
properter = QgsDistanceArcProperter()
```

Apoi spunei directorului despre această strategie

```
director.addProperter(properter)
```

Acum putem crea constructorul, care va crea graful. Constructorul clasei `QgsGraphBuilder` ia mai multe argumente:

- `crs` — sistemul de coordonate de referință de utilizat. Argument obligatoriu.

- `otfEnabled` — utilizezi sau nu reproiectarea “din zbor”. În mod implicit `const:True` (folosii OTF).
- `topologyTolerance` — toleranța topologică. Valoarea implicită este 0.
- `ellipsoidID` — elipsoidul de utilizat. În mod implicit “WGS84”.

```
# only CRS is set, all other values are defaults
builder = QgsGraphBuilder(myCRS)
```

De asemenea, putem defini mai multe puncte, care vor fi utilizate în analiză. De exemplu

```
startPoint = QgsPoint(82.7112, 55.1672)
endPoint = QgsPoint(83.1879, 54.7079)
```

Acum că totul este la locul lui, putem să construim graful și să “legăm” aceste puncte la el

```
tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
```

Construirea unui graf poate dura ceva timp (depinzând de numărul de entități dintr-un strat și de dimensiunea stratului). `tiedPoints` reprezintă o listă cu coordonatele punctelor “asociate”. Când s-a terminat operațiunea de construire putem obține graful și să-l utilizăm pentru analiză

```
graph = builder.graph()
```

Cu următorul cod putem obține indecii punctelor noastre

```
startId = graph.findVertex(tiedPoints[0])
endId = graph.findVertex(tiedPoints[1])
```

19.3 Analiza grafului

Analiza de reea este utilizată pentru a găsi răspunsuri la două întrebări: care noduri sunt conectate și identificarea celei mai scurte căi. Pentru a rezolva această problemă, biblioteca de analiză de reea oferă algoritmul lui Dijkstra.

Algoritmul lui Dijkstra găsește cea mai bună cale între unul dintre vârfurile grafului și toate celelalte, precum și valorile parametrilor de optimizare. Rezultatele pot fi reprezentate ca cel mai scurt arbore.

Arborele drumurilor cele mai scurte reprezintă un graf (sau mai precis — arbore) orientat, ponderat, cu următoarele proprietăți:

- doar un singur nod nu are muchii de intrare — rădăcina arborelui
- toate celelalte noduri au numai o margine de intrare
- dacă nodul B este accesibil din nodul A, apoi calea de la A la B este singura disponibilă și este optimă (cea mai scurtă) în acest graf

Pentru a obține cel mai scurt arbore folosiți metodele `shortestTree()` și `dijkstra()` ale clasei `QgsGraphAnalyzer`. Se recomandă utilizarea metodei `dijkstra()`, deoarece lucrează mai rapid și utilizează memoria mult mai eficient.

Metoda `shortestTree()` este utilă atunci când doriți să vă plimbi de-a lungul celei mai scurte căi. Aceasta creează mereu un nou obiect de tip graf (`QgsGraph`) care acceptă trei variabile:

- `source` — graf de intrare
- `startVertexIdx` — Indexul punctului de pe arbore (rădăcina arborelui)
- `criterionNum` — numărul de proprietăți marginii de folosit (începând de la 0).

```
tree = QgsGraphAnalyzer.shortestTree(graph, startId, 0)
```

The `dijkstra()` method has the same arguments, but returns two arrays. In the first array element `i` contains index of the incoming edge or -1 if there are no incoming edges. In the second array element `i` contains distance from the root of the tree to vertex `i` or `DOUBLE_MAX` if vertex `i` is unreachable from the root.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, startId, 0)
```

Iată un cod foarte simplu pentru a afla arborele celei mai scurte căi, folosind graful creat cu metoda `shortestTree()` (selectai stratul linie în TOC și înlocuiește coordonatele cu ale dvs). **Atenție:** folosește acest cod doar ca exemplu, deoarece el va crea o mulțime de obiecte `QgsRubberBand`, putând fi lent pe seturi de date de mari dimensiuni.

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.743804, 0.22954)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)

i = 0;
while (i < tree.arcCount()):
    rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
    rb.setColor (Qt.red)
    rb.addPoint (tree.vertex(tree.arc(i).inVertex()).point())
    rb.addPoint (tree.vertex(tree.arc(i).outVertex()).point())
    i = i + 1
```

Acelai lucru, dar cu ajutorul metodei `dijkstra()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-1.37144, 0.543836)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)
```

```
(tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

for edgeId in tree:
    if edgeId == -1:
        continue
    rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
    rb.setColor(Qt.red)
    rb.addPoint(graph.vertex(graph.arc(edgeId).inVertex()).point())
    rb.addPoint(graph.vertex(graph.arc(edgeId).outVertex()).point())
```

19.3.1 Găsirea celor mai scurte căi

Pentru a găsi calea optimă între două puncte este utilizată următoarea abordare. Ambele puncte (se începe cu A și se termină cu B) sunt “legate” de un graf, atunci când se construiește. Apoi folosind metodele `shortestTree()` sau `dijkstra()` vom construi cel mai scurt arbore cu rădăcina în punctul de pornire A. În acel arbore am găsit, de asemenea, punctul de final B și începem parcurgerea arborelui de la punctul B la punctul A. Întregul algoritm poate fi scris ca

```
assign = B
while != A
    add point to path
    get incoming edge for point
    look for point, that is start point of this edge
    assign =
add point to path
```

În acest moment avem calea, sub formă de listă inversată de noduri (nodurile sunt listate în ordine inversă, de la punctul de final către cel de start), ele fiind vizitate în timpul parcurgerii căii.

Aici este codul de test pentru consola Python a QGIS (va trebui să selectai stratul linie în TOC și să înlocuiești coordonatele din cod cu ale dvs.), care folosește metoda `shortestTree()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.835953, 0.15679)
pStop = QgsPoint(-1.1027, 0.699986)

tiedPoints = director.makeGraph(builder, [pStart, pStop])
graph = builder.graph()

tStart = tiedPoints[0]
tStop = tiedPoints[1]

idStart = graph.findVertex(tStart)
tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)

idStart = tree.findVertex(tStart)
idStop = tree.findVertex(tStop)
```

```
if idStop == -1:
    print "Path not found"
else:
    p = []
    while (idStart != idStop):
        l = tree.vertex(idStop).inArc()
        if len(l) == 0:
            break
        e = tree.arc(l[0])
        p.insert(0, tree.vertex(e.inVertex()).point())
        idStop = e.outVertex()

    p.insert(0, tStart)
    rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
    rb.setColor(Qt.red)

    for pnt in p:
        rb.addPoint(pnt)
```

Iar aici este același exemplu, dar folosind metoda dijkstra()

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.835953, 0.15679)
pStop = QgsPoint(-1.1027, 0.699986)

tiedPoints = director.makeGraph(builder, [pStart, pStop])
graph = builder.graph()

tStart = tiedPoints[0]
tStop = tiedPoints[1]

idStart = graph.findVertex(tStart)
idStop = graph.findVertex(tStop)

(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

if tree[idStop] == -1:
    print "Path not found"
else:
    p = []
    curPos = idStop
    while curPos != idStart:
        p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
        curPos = graph.arc(tree[curPos]).outVertex();

    p.append(tStart)

    rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
    rb.setColor(Qt.red)
```



```
for pnt in p:
    rb.addPoint(pnt)
```

19.3.2 Ariile de disponibilitate

Aria de disponibilitate a nodului A este un subset de noduri ale graf-ului, care sunt accesibile din nodul A iar costurile căii de la A la aceste noduri nu sunt mai mari decât o anumită valoare.

Mai clar, acest lucru poate fi dovedit cu următorul exemplu: “Există o echipă de intervenție în caz de incendiu. Ce zone ale orașului acoperă această echipă în 5 minute? Dar în 10 minute? Dar în 15 minute?”. Răspunsul la aceste întrebări îl reprezintă zonele de disponibilitate ale echipei de intervenție.

Pentru a găsi zonele de disponibilitate putem folosi metoda `dijkstra()` a clasei `QgsGraphAnalyzer`. Este suficientă compararea elementelor matricei de costuri cu valoarea predefinită. În cazul în care costul[i] este mai mic sau egal decât o valoare predefinită, atunci nodul i se află în zona de disponibilitate, în caz contrar este în afară.

Mai dificilă este obținerea granielor zonei de disponibilitate. Marginea de jos reprezintă un set de noduri care încă sunt accesibile, iar marginea de sus un set de noduri inaccesibile. De fapt, acest lucru este simplu: marginea disponibilă a atins aceste margini parcurgând arborele cel mai scurt, pentru care nodul de start este accesibil, spre deosebire de celelalte capăt, care nu este accesibil.

Iată un exemplu

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(65.5462, 57.1509)
delta = qgis.utils.iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
rb.setColor(Qt.green)
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() + delta))

tiedPoints = director.makeGraph(builder, [pStart])
graph = builder.graph()
tStart = tiedPoints[0]

idStart = graph.findVertex(tStart)

(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

upperBound = []
r = 2000.0
i = 0
while i < len(cost):
    if cost[i] > r and tree[i] != -1:
        outVertexId = graph.arc(tree[i]).outVertex()
```

```
    if cost[outVertexId] < r:
        upperBound.append(i)
    i = i + 1

for i in upperBound:
    centerPoint = graph.vertex(i).point()
    rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
    rb.setColor(Qt.red)
    rb.addPoint(QgsPoint(centerPoint.x() - delta, centerPoint.y() - delta))
    rb.addPoint(QgsPoint(centerPoint.x() + delta, centerPoint.y() - delta))
    rb.addPoint(QgsPoint(centerPoint.x() + delta, centerPoint.y() + delta))
    rb.addPoint(QgsPoint(centerPoint.x() - delta, centerPoint.y() + delta))
```

-
- în execuție
 - aplicații personalizate, 3
 - încărcare
 - Fiiere GPX, 8
 - Geometriei MySQL, 8
 - proiecte, 5
 - straturi cu text delimitat, 7
 - Straturi OGR, 7
 - Straturi PostGIS, 7
 - straturi raster, 8
 - Straturi SpatiaLite, 8
 - straturi vectoriale, 7
 - Straturi WMS, 9
 - API, 1
 - aplicații personalizate
 - în execuție, 3
 - Python, 2
 - atribute
 - straturi vectoriale entități, 15
 - calcularea valorilor, 46
 - consolă
 - Python, 2
 - entități
 - atribute, straturi vectoriale, 15
 - straturi vectoriale iterarea, 15
 - straturi vectoriale selecție, 15
 - expresii, 46
 - evaluare, 48
 - parsare, 47
 - Fiiere GPX
 - încărcare, 8
 - filtrare, 46
 - furnizor de memorie, 21
 - geometrie
 - accesare, 31
 - construire, 31
 - manipulare, 29
 - predicată și operațiuni, 32
 - Geometriei MySQL
 - încărcare, 8
 - ieire
 - folosirea Compozitorului de Hări, 44
 - imagine raster, 45
 - PDF, 46
 - index spațial
 - folosind, 19
 - inițializare
 - Python, 1
 - interogare
 - straturi raster, 13
 - iterarea
 - entități, straturi vectoriale, 15
 - mediu
 - PYQGIS_STARTUP, 1
 - metadata, 60
 - metadata.txt, 60
 - personalizat
 - rendere, 27
 - plugin-uri, 73
 - apelarea unei metode printr-o combinație rapidă de taste, 77
 - atributele de acces ale entităților selectate, 77
 - comutarea straturilor, 77
 - depozitul oficial al plugin-urilor python, 74
 - dezvoltare, 55
 - documentație, 62
 - fier de resurse, 62
 - fragmente de cod, 62
 - implementare help, 62
 - lansarea, 68
 - metadata.txt, 58, 60
 - scriere, 57
 - scriere cod, 58
 - testare, 68
 - proiecții, 36
 - proiecte
 - încărcare, 5
 - PYQGIS_STARTUP
 - mediu, 1
 - Python
 - aplicații personalizate, 2
 - consolă, 2
 - dezvoltarea plugin-urilor, 55
-

- inițializare, 1
- plugin-uri, 2
- startup.py, 1
- randare hartă, 42
 - simplu, 43
- rastere
 - multibandă, 13
 - simplă bandă, 12
- recitare
 - straturi raster, 13
- registru straturilor de hartă, 9
 - adăugarea unui strat, 9
- render cu simbol gradual, 24
- render cu simbologie clasificată, 23
- render cu un singur simbol , 23
- rendere
 - personalizat, 27
- resources.qrc, 62
- selecție
 - entități, straturi vectoriale, 15
- setări
 - citire, 49
 - global, 51
 - proiect, 51
 - stocare, 49
 - strat de hartă, 52
- simbologie
 - render cu simbol clasificat, 23
 - render cu simbol gradual, 24
 - render cu un singur simbol , 23
 - vechi, 29
- simboluri
 - lucrul cu, 24
- sisteme de coordonate de referință, 35
- startup.py
 - Python, 1
- straturi cu text delimitat
 - încărcare, 7
- Straturi OGR
 - încărcare, 7
- Straturi PostGIS
 - încărcare, 7
- straturi raster
 - încărcare, 8
 - detalii, 11
 - drawing style, 11
 - folosind, 9
 - interogare, 13
 - recitare, 13
- Straturi SpatialLite
 - încărcare, 8
- straturi vectoriale
 - încărcare, 7
 - editare, 17
 - entități atribute, 15
 - iterarea entității, 15
 - scris, 20
 - selecție entități, 15
 - simbologie, 22
- Straturi WMS
 - încărcare, 9
- straturile plugin-ului, 68
 - subclasarea QgsPluginLayer, 69
- straturile simbolului
 - crearea tipurilor personalizate, 25
 - lucrul cu, 25
- suportul hărții, 36
 - încapsulare, 37
 - arhitectură, 37
 - benzi de cauciuc, 39
 - dezvoltarea elementelor personalizate pentru suportul de hartă , 41
 - dezvoltarea instrumentelor de hartă personalizate, 40
 - instrumente pentru hartă, 38
 - marcaje vertex, 39
- tipărire hartă, 42