
PyQGIS developer cookbook

Release 2.2

QGIS Project

December 04, 2014

1	Introducere	1
1.1	Consola Python	1
1.2	Plugin-uri Python	2
1.3	Aplicaii	2
2	Încărcarea Straturilor	5
2.1	Straturi Vectoriale	5
2.2	Straturi raster	6
2.3	Registrul straturilor de hartă	7
3	Utilizarea straturilor raster	9
3.1	Detaliile stratului	9
3.2	Stilul desenării	9
3.3	Recitirea straturilor	11
3.4	Interogarea valorilor	11
4	Utilizarea straturilor vectoriale	13
4.1	Iteraii în straturile vectoriale	13
4.2	Modificarea straturilor vectoriale	14
4.3	Modificarea straturi vectoriale prin editarea unui tampon de memorie	15
4.4	Crearea unui index spaial	16
4.5	Scrierea straturilor vectoriale	17
4.6	Furnizorul de memorie	18
4.7	Aspectul (simbologia) straturilor vectoriale	19
5	Manipularea geometriei	27
5.1	Construirea geometriei	27
5.2	Accesarea geometriei	27
5.3	Predicate i operaiuni geometrice	28
6	Proiecii suportate	29
6.1	Sisteme de coordonate de referină	29
6.2	Proiecii	30
7	Folosirea suportului de hartă	31
7.1	Încapsularea suportului de hartă	31
7.2	Folosirea instrumentelor în suportul de hartă	32
7.3	Benzile elastice i marcajele nodurilor	33
7.4	Dezvoltarea instrumentelor personalizate pentru suportul de hartă	34
7.5	Dezvoltarea elementelor personalizate pentru suportul de hartă	35
8	Randarea hărților i imprimarea	37
8.1	Randarea simplă	37

8.2	Generarea folosind Compozitorul de hări	38
9	Expresii, filtrarea și calculul valorilor	41
9.1	Parsarea expresiilor	42
9.2	Evaluarea expresiilor	42
9.3	Exemple	42
10	Citirea și stocarea setărilor	45
11	Comunicarea cu utilizatorul	47
11.1	Showing messages. The QgsMessageBar class.	47
11.2	Afiarea progresului	50
11.3	Jurnalizare	50
12	Dezvoltarea plugin-urilor Python	53
12.1	Scrierea unui plugin	53
12.2	Conținutul Plugin-ului	54
12.3	Documentație	58
13	Setările IDE pentru scrierea și depanarea de plugin-uri	59
13.1	O notă privind configurarea IDE-ului în Windows	59
13.2	Depanare cu ajutorul Eclipse și PyDev	60
13.3	Depanarea cu ajutorul PDB	64
14	Utilizarea straturilor plugin-ului	65
14.1	Subclasarea QgsPluginLayer	65
15	Compatibilitatea cu versiunile QGIS anterioare	67
15.1	Meniul plugin-ului	67
16	Lansarea plugin-ului dvs.	69
16.1	Depozitul oficial al plugin-urilor python	69
17	Fragmente de cod	71
17.1	Cum să apelăm o metodă printr-o combinație rapidă de taste	71
17.2	How to toggle Layers (work around)	71
17.3	Cum să accesezi tabelul de atribute al entităților selectate	71
18	Biblioteca de analiză a rețelelor	73
18.1	Informații generale	73
18.2	Building graph	73
18.3	Analiza grafului	75
	Index	81

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 funcionalității 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 cu Qt (PyQt) se face, de asemenea, cu ajutorul SIP, acest lucru permiând integrarea perfectă a PyQGIS cu PyQt.

DE EFECTUAT: Noiuni de bază PyQGIS (Compilare manuală, Depanare)

Există mai multe modalități de a crea legături între QGIS i Python, acestea fiind detaliate în următoarele secțiuni:

- 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 identic 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 Consola Python

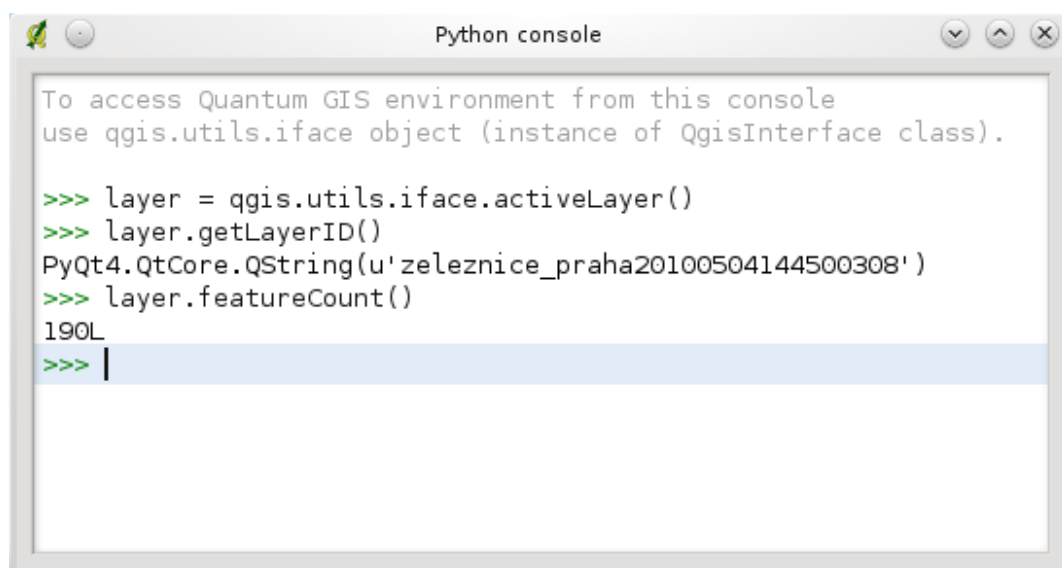
Pentru scripting, se poate utiliza consola Python integrată. Aceasta poate fi deschisă din meniul: *Plugins* → *Consola Python*. Consola se deschide ca o fereastră de utilități non-modală:

Captura de ecran de mai sus ilustrează cum să obinei accesul la stratul curent selectat în lista straturilor, să-i afișezi ID-ul i, opțional, în cazul în care acesta este un strat vectorial, să calculezi numărul total de entități spațiale. Pentru interacțiunea cu mediul QGIS, există o variabilă de :date: *iface*, care este o instanță a :clasei: *QgsInterface*. Această interfață permite accesul la suprafața hărții, meniuri, barele de instrumente i la alte părți ale aplicației QGIS.

For convenience of the user, the following statements are executed when the console is started (in future it will be possible to set further initial commands):

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

Pentru cei care folosesc des consola, ar putea fi util să stabilească o comandă rapidă pentru declanșarea consolei (în meniul :menuselection: *Settings* -> *Configurare comenzi rapide* ...)



```

Python console

To access Quantum GIS environment from this console
use qgis.utils.iface object (instance of QgisInterface class).

>>> layer = qgis.utils.iface.activeLayer()
>>> layer.getLayerID()
PyQt4.QtCore.QString(u'zeleznice_praha20100504144500308' )
>>> layer.featureCount()
190L
>>> |

```

Figura 1.1: Consola Python din QGIS

1.2 Plugin-uri Python

QGIS permite îmbunătățirea funcționalității sale, folosind plugin-uri. Acest lucru a fost inițial posibil numai cu ajutorul limbajului C. Odată cu adăugarea în QGIS a suportului pentru Python, este posibilă și folosirea de plugin-uri scrise în Python. Principalul avantaj față de plugin-urile în C este simplitatea distribuției (nu este necesară compilarea pentru fiecare platformă) iar dezvoltarea este mai ușoară.

Multe plugin-uri care acoperă diverse funcționalități au fost scrise de la introducerea suportului pentru Python. Instalatorul de plugin-uri permite utilizatorilor aducerea cu ușurință, actualizarea și eliminarea plugin-urilor Python. A se vedea pagina [Depozitele de Plugin-uri Python](#) pentru diferitele surse de plugin-uri.

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

1.3 Aplicații

Adesea, atunci când are loc procesarea unor date GIS, este recomandabilă crearea unor script-uri pentru automatizarea procesului în loc de a face iarăși și iarăși aceeași sarcină. Cu PyQGIS, acest lucru este perfect posibil — importai `:modul: qgis.core`, îl inițializezi și sunteți gata pentru prelucrare.

Sau poate doriți să creați o aplicație interactivă care utilizează unele funcționalități GIS — măsurarea anumitor date, exportarea unei hărți în format PDF sau orice alte funcționalități. Modulul `qgis.gui` aduce în plus diverse componente GUI, mai ales widget-ul ariei hărții, care poate fi foarte ușor încorporat în aplicațiile cu suport pentru zoom, panning și/sau orice alte instrumente personalizabile.

1.3.1 Utilizarea PyQGIS în aplicații personalizate

Notă: *nu* utilizai `qgis.py` ca nume pentru script-ul de test — datorită acestui nume, Python nu va fi capabil să importe legăturile.

Mai întâi de toate, trebuie să importai modul `qgis`, să setezi calea către resurse — baza de date a proiecțiilor, furnizorii etc. Când setezi prefixul căii având ca al doilea argument `:const: True`, QGIS va inițializa toate căile cu standardul `dir` în directorul prefixului. La apelarea funcției `initQgis()` este important să permiți aplicației QGIS să caute furnizorii disponibili.

```
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 QGIS — încărcați straturile și faceți unele prelucrări sau startați un GUI cu o zonă pentru o hartă. Posibilitățile sunt nelimitate :-)

Când ai terminat cu utilizarea bibliotecii QGIS, apelei funcția `exitQgis()` pentru a vă asigura că totul este curat (de exemplu, curățați registrul stratului hărții și tergeți straturile):

```
QgsApplication.exitQgis()
```

1.3.2 Rularea aplicațiilor personalizate

Trebuie să indicați sistemului dvs. unde să caute bibliotecile QGIS și modulele Python corespunzătoare, dacă acestea nu sunt într-o locație standard — altfel Python va semnaliza:

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

Aceasta se poate repara prin setarea variabilei de mediu `PYTHONPATH`. În următoarele comenzi, `qgispath` ar trebui să fie înlocuit de către calea de instalare actuală a QGIS:

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

Calea către modulele PyQGIS este acum cunoscută, totuși, acestea depind de bibliotecile `qgis_core` și `qgis_gui` (modulele Python servesc numai pentru ambalare). Calea către aceste biblioteci este de obicei necunoscută sistemului de operare, astfel încât vei obține iarăși o eroare de import (mesajul poate 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
```

Remediați acest lucru prin adăugarea directoarelor în care rezidă bibliotecile QGIS la calea de căutare a linker-ului dinamic:

- 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 va avea grijă de pornire. Când livrați aplicații personalizate folosind PyQGIS, există, de obicei, două posibilități:

- să cereți utilizatorului să instaleze QGIS pe platforma sa înainte de a instala aplicația. Instalatorul aplicației ar trebui să caute locațiile implicite ale bibliotecilor QGIS și să permită utilizatorului setarea căii, în cazul în care nu poate fi găsită. Această abordare are avantajul de a fi mai simplă, cu toate acestea este nevoie ca utilizatorul să parcurgă mai multe etape.
- să împacheteze QGIS împreună cu aplicația dumneavoastră. Livrarea aplicației poate fi mai dificilă, iar pachetul va fi mai mare, dar utilizatorul va fi salvat de povara de a descărca și instala software suplimentar.

Cele două modele pot fi combinate - pui distribuții aplicații independente pe Windows și Mac OS X, lăsând la îndemâna utilizatorului și a managerul de pachete instalarea QGIS pe Linux.

Încărcarea Straturilor

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

2.1 Straturi Vectoriale

Pentru a încărca un strat vectorial, specificai identificatorul sursei de date a stratului, numele stratului și numele furnizorului:

```
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 invalid.

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

- Bibliotecă OGR (fieri shape și multe alte formate de fiere) — sursa de date este calea către fier

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

- Bază de date PostGIS — sursa de date este un ir cu toate informațiile necesare pentru a crea o conexiune la baza de date PostgreSQL. Clasa `QgsDataSourceURI` poate genera acest ir pentru dvs. Rețineți că QGIS trebuie să fie compilat cu suport Postgres, în caz contrar acest furnizor nu va fi disponibil.

```
uri = QgsDataSourceURI()
# set host name, port, database name, username and password
uri.setConnection("localhost", "5432", "dbname", "johny", "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 sau alte fiere text delimitate — pentru a deschide un fier având punct și virgulă ca delimitator, iar câmpurile “x” și “y” ca și coordonate x respectiv y, ar trebui să folosești ceva de genul următor

```
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")
```

- Fiiere GPX — furnizorul de date “gpx” citete urme, rute i puncte de referină din fiere gpx. Pentru a deschide un fier, tipul (urmă / traseu / punct de referină) trebuie să fie specificat ca parte a url-ului

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

- Bază de date SpatialLite — începând cu QGIS v1.1. În mod similar bazelor de date PostGIS, QgsDataSourceURI poate fi utilizat pentru generarea identificatorului sursei de date

```
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')
```

- Geometrii MySQL bazate pe WKB, prin OGR — sursa de date este irul de conectare la tabelă

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

- Conexiune WFS: conexiunea este definită cu un URI i folosind furnizorul “WFS”

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

URI poate fi creat folosind biblioteca standard urllib.

```
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))
```

And you can also use the

2.2 Straturi raster

Pentru accesarea fiierelor raster este utilizată biblioteca GDAL. Acesta suportă o gamă largă de formate de fiere. În cazul în care avei probleme cu deschiderea unor fiere, verificai dacă GDAL are suport pentru formatul respectiv (nu toate formatele sunt disponibile în mod implicit). Pentru a încărca un raster dintr-un fier, specificai numele fierului i numele de bază

```
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!"
```

Straturile raster pot fi, de asemenea, create dintr-un serviciu WCS.

```
layer_name = 'elevation'
uri = QgsDataSourceURI()
uri.setParam ('url', 'http://localhost:8080/geoserver/wcs')
uri.setParam ( "identifier", layer_name)
rlayer = QgsRasterLayer(uri, 'my_wcs_layer', 'wcs')
```

Alternativ, putei încărca un strat raster de pe un server WMS. Cu toate acestea, în prezent, nu este posibilă accesarea răspunsului GetCapabilities de la API — trebuie să cunoști straturile dorite

```
urlWithParams = 'url=http://wms.jpl.nasa.gov/wms.cgi&layers=global_mosaic&\
styles=pseudo&format=image/jpeg&crs=EPSG:4326'
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
    print "Layer failed to load!"
```

2.3 Registrul straturilor de hartă

Dacă dorești să utilizezi straturile deschise pentru randare, nu uitați să le adăugați la 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.

Adăugarea unui strat la registru:

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

Straturile sunt distruse în mod automat la ieșire; cu toate acestea, dacă dorești să tergeți stratul în mod explicit, atunci folosește:

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

DE EFECTUAT: Mai multe despre registrul straturilor de hartă?

Utilizarea straturilor raster

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

3.1 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 făcând referire la culori stocate în paletă.

```
>>> rlayer.width(), rlayer.height()
(812, 301)
>>> rlayer.extent()
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
```

3.2 Stilul desenării

Când un strat raster este încărcat, în funcție de tipul său, va moteni un stil de desenare implicit. Acesta poate fi modificat, fie prin modificarea manuală a proprietăților rasterului, fie programatic. Există următoarele stiluri de desenare:

In-dex	Constant: QgsRasterLater.X	Comentariu
1	SingleBandGray	Imagine cu o singură bandă, afiată într-o gamă de nuane de gri
2	SingleBandPseudo-Color	Imagine cu bandă unică, desenată de un algoritm cu pseudoculori
3	PalettedColor	“Palette” image drawn using color table
4	PalettedSingle-BandGray	“Palette” layer drawn in gray scale
5	PalettedSingle-BandPseudoColor	“Palette” layer drawn using a pseudocolor algorithm
7	MultiBandSingle-BandGray	Strat care conține 2 sau mai multe benzi, din care o singură bandă desenată într-o gamă cu tonuri de gri
8	MultiBandSingle-BandPseudoColor	Strat care conține 2 sau mai multe benzi, din care o singură bandă este desenată folosind un algoritm cu pseudoculori
9	MultiBandColor	Strat conținând 2 sau mai multe benzi, mapate în spațiul de culori RGB.

Pentru a interoga stilul de desenare curent:

```
>>> rlayer.drawingStyle()
9
```

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 pseudoculori, 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 pseudoculori.

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

TODO: contrast enhancements, transparency (no data), user defined min/max, band statistics

3.2.1 Rastere cu o singură bandă

Sunt redat în nuane de gri, în mod implicit. Pentru a se schimba stilul de desenare pentru pseudoculori:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandPseudoColor)
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
```

The `PseudoColorShader` is a basic shader that highlights low values in blue and high values in red. Another, `FreakOutShader` uses more fancy colors and according to the documentation, it will frighten your granny and make your dogs howl.

Există, de asemenea, `ColorRampShader`, care mapează culorile specificate după propria hartă de culori. Dispune de trei moduri de interpolare a valorilor:

- 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

Pentru a seta o gamă de culori pentru umbrire interpolate, variind de la verde la galben (pentru valori ale pixelilor între 0-255):

```
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.ColorRampShader)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
          QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn = rlayer.rasterShader().rasterShaderFunction()
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

Pentru a reveni la nuanțele de gri implicite, utilizați:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandGray)
```

3.2.2 Rastere multibandă

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

```
>>> rlayer.setGreenBandName(rlayer.bandName(1))
>>> rlayer.setRedBandName(rlayer.bandName(2))
```

În cazul în care doar o singură bandă este necesară pentru vizualizarea rasterului, poate fi aleasă desenarea simplă bandă — fie în tonuri de gri fie cu pseudocolori, ca în seciunea anterioară:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.MultiBandSingleBandPseudoColor)
>>> rlayer.setGrayBandName(rlayer.bandName(1))
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
>>> # now set the shader
```

3.3 Recitirea straturilor

Dacă schimbi simbologia stratului i ai dori 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 funcționalitate este disponibilă începând de la QGIS 1.4, în versiunile anterioare această funcție neexistând — pentru a fi siguri de cod, că funcționează cu toate versiunile de QGIS, vom verifica în primul rând dacă metoda există.

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

With WMS raster layers, these commands do not work. In this case, you have to do it explicitly:

```
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 forța QGIS să actualizeze simbologia din lista straturilor (legendă). Acest lucru poate fi realizat după cum urmează (`iface` este un exemplu de `QgisInterface`)

```
iface.legendInterface().refreshLayerSymbology(layer)
```

3.4 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()
```

The `results` method in this case returns a dictionary, with band indices as keys, and band values as values.

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

Utilizarea straturilor vectoriale

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

4.1 Iterații în straturile vectoriale

Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. the `layer` variable is assumed to have a `QgsVectorLayer` object

```
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
```

Attributes can be referred by index.

```
idx = layer.fieldNameIndex('name')
print feature.attributes()[idx]
```

4.1.1 Parcurgerea entităților selectate

4.1.2 Convenience methods

For the above cases, and in case you need to consider selection in a vector layer in case it exist, you can use the `features()` method from the built-in processing plugin, as follows:

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

În acest mod se vor parcurge toate entitățile stratului, în cazul în care nu există o selecție, sau, în caz contrar, doar entitățile selectate.

dacă aveți nevoie doar de caracteristicile selectate, puteți utiliza metoda `selectedFeatures()` a stratului vectorial:

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

4.1.3 Parcurgerea unui subset de entități

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

```
request=QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for f in layer.getFeatures(request):
    ...
```

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

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

4.2 Modificarea straturilor vectoriale

Most vector data providers support editing of layer data. Sometimes they support just a subset of possible editing actions. Use the `capabilities()` function to find out what set of functionality is supported:

```
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 doriți să faceți doar schimbări temporare, treceți la secțiunea următoare, care explică modul de editare al unei memorii tampon `<editing-buffer>`.

4.2.1 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 ] )

```

4.2.2 tergerea entităților

To delete some features, just provide a list of their feature IDs:

```

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

```

4.2.3 Modificarea entităților

It is possible to either change feature's geometry or to change some attributes. The following example first changes values of attributes with index 0 and 1, then it changes the feature's geometry:

```

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 })

```

4.2.4 Adăugarea i eliminarea câmpurilor

To add fields (attributes), you need to specify a list of field definitions. For deletion of fields just provide a list of field indexes.

```

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 schimbările nu se propagă în mod automat.

```
layer.updateFields()
```

4.3 Modificarea straturi vectoriale prin editarea unui tampon de memorie

Când editai vectori în aplicaia QGIS, în primul rând, trebuie să comuți în modul de editare pentru stratul în care lucrezi, 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 salvei 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)
```

In order to make undo/redo work properly, the above mentioned calls have to be wrapped into undo commands. (If you do not care about undo/redo and want to have the changes stored immediately, then you will have easier work by *editing with data provider*.) How to use the undo functionality

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

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

if problem_occurred:
    layer.destroyEditCommand()
    return

# ... more editing ...

layer.endEditCommand()
```

The `beginEndCommand()` will create an internal “active” command and will record subsequent changes in vector layer. With the call to `endEditCommand()` the command is pushed onto the undo stack and the user will be able to undo/redo it from GUI. In case something went wrong while doing the changes, the `destroyEditCommand()` method will remove the command and rollback all changes done while this command was active.

Pentru a activa modul de editare, este disponibilă metoda `startEditing()`, pentru a opri editarea există `func:commitChanges` și `rollback()` — totuși, în mod normal, ar trebui să nu nevoie de aceste metode și să permiteți utilizatorului declanșarea acestor funcționalități.

4.4 Crearea unui index spațial

Spatial indexes can dramatically improve the performance of your code if you need to do frequent queries to a vector layer. Imagin, for instance, that you are writing an interpolation algorithm, and that for a given location you need to know the 10 closest point from a points layer, in order to use those point for calculating the interpolated value. Without a spatial index, the only way for QGIS to find those 10 points is to compute the distance from each and every point to the specified location and then compare those distances. This can be a very time consuming task, specially if it needs to be repeated from several locations. If a spatial index exists for the layer, the operation is much more effective.

Think of a layer without a spatial index as a telephone book in which telephone numbers are not ordered or indexed. The only way to find the telephone number of a given person is to read from the beginning until you find it.

Indicii spațiali nu sunt creați în mod implicit pentru un strat vectorial QGIS, dar îi puteți realiza cu ușurință. Iată ce trebuie să faceți.

1. create spatial index — the following code creates an empty index:

```
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(feats)
```

3. once spatial index is filled with some values, you can do some queries:

```
# 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))
```

4.5 Scrierea straturilor vectoriale

Putei scrie în fierele coninând straturi vectoriale folosind clasa `QgsVectorFileWriter`. Aceasta acceptă orice alt tip de fier vector care suportă OGR (fiere shape, GeoJSON, KML i altele).

Există două posibilități de a exporta un strat vectorial:

- from an instance of `QgsVectorLayer`:

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

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

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

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

Al treilea parametru se referă la codificarea textului de ieire. Doar unele formate au nevoie de acest lucru pentru o funcționare corectă - fiierul shape fiind printre ele — totui, în cazul în care nu utilizezi caractere internaionale nu trebuie să vă îngrijoreze codificarea. În al patrulea parametru, care acum are valoarea `None`, se poate specifica destinaia CRS — dacă este trecută o instanță validă a `QgsCoordinateReferenceSystem`, stratul este transformat pentru acel 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.

- directly from features:

```
# 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 WKBTYP enum
# 5. layer's spatial reference (instance of
#   QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
```

```
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, \
    Qgs.WKBPoint, None, "ESRI Shapefile")

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
```

4.6 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 capabili 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=definiție 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.

The following example of a URI incorporates all these options:

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

The following example code illustrates creating and populating a memory provider:

```
# 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) ] )

# 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()

```

Finally, let's check whether everything went well:

```

# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMin(), e.yMin(), e.xMax(), e.yMax()

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

```

4.7 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 grijă 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()
```

And with that reference, let us explore it a bit:

```
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 să nu presupunem că există numai aceste tipuri. Puteți interoga singleton-ul `QgsRendererV2Registry` pentru a afla tipurile de randare disponibile în prezent.

It is possible to obtain a dump of a renderer contents in text form — can be useful for debugging:

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

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

To get a list of categories:

```

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()`.

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.

To find out more about ranges used in the renderer:

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

puteți 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.

If you wish to create your own graduated symbol renderer you can do so as illustrated in the example snippet below (which creates a simple two class arrangement):

```
from qgis.core import (QgsVectorLayer,
                      QgsMapLayerRegistry,
                      QgsGraduatedSymbolRendererV2,
                      QgsSymbolV2,
                      QgsRendererRangeV2)

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)

```

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

- `QgsMarkerSymbolV2` - for point features
- `QgsLineSymbolV2` - for line features
- `QgsFillSymbolV2` - for polygon features

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.

Having an instance of a symbol (e.g. from a renderer), it is possible to explore it: `type()` method says whether it is a marker, line or fill symbol. There is a `dump()` method which returns a brief description of the symbol. To get a list of symbol layers:

```

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, putei interoga pentru dimensiunea simbolului `i` unghiul de rotaie cu metodele `size()` `i` `angle()`, iar pentru simbolurile linie există metoda `width()` care returnează lăimea liniei.

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

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 `i`, 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 `i` implicite sunt `SimpleMarker`, `SimpleLine` `i` `SimpleFill`.

You can get a complete list of the types of symbol layers you can create for a given symbol layer class like this:

```

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

```

Output:

```

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 valoriceie 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 `i` unghiul sunt disponibile doar pentru straturi simbol de tip marcer iar lăimea pentru straturi simbol de tip linie.

Imagine you would like to customize the way how the data gets rendered. You can create your own symbol layer class that will draw the features exactly as you wish. Here is an example of a marker that draws red circles with specified radius:

```
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.

Usually it is convenient to add a GUI for setting attributes of the symbol layer type to allow users to customize the appearance: in case of our example above we can let user set circle radius. The following code implements such 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 aloca 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 conștient 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.

We will have to create metadata for the symbol layer:

```

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 transmiteți 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 — i am încheiat.

Ar putea fi utilă crearea unei noi implementări de render, dacă doriți să personalizați 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

The following code shows a simple custom renderer that creates two marker symbols and chooses randomly one of them for every feature:

```

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(QGis.Point) ]

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 o ateață renderul. În cele din urmă `clone()` ar trebui să returneze o copie a renderului.

Like with symbol layers, it is possible to attach a GUI for configuration of the renderer. It has to be derived from `QgsRendererV2Widget`. The following sample code creates a button that allows user to set symbol of the first symbol:

```
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.

The last missing bit is the renderer metadata and registration in registry, otherwise loading of layers with the renderer will not work and user will not be able to select it from the list of renderers. Let us finish our `RandomRenderer` example:

```

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 metadate abstracte așteaptă numele renderului, nume vizibil pentru utilizatori și numele opțional 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.

To associate an icon with the renderer you can assign it in `QgsRendererV2AbstractMetadata` constructor as a third (optional) argument — the base class constructor in the `RandomRendererMetadata` `__init__()` function becomes:

```

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 metadate. 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).

re **TODO:**

- creating/modifying symbols
- working with style (`QgsStyleV2`)
- working with color ramps (`QgsVectorColorRampV2`)
- rule-based renderer (see .. [this](http://snorf.net/blog/2014/03/04/symbology-of-vector-layers-in-qgis-python-plugins) blogpost: <http://snorf.net/blog/2014/03/04/symbology-of-vector-layers-in-qgis-python-plugins>)
- exploring symbol layer and renderer registries

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.

5.1 Construirea geometriei

Există mai multe opțiuni pentru a crea o geometrie:

- from coordinates:

```
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.

- from well-known text (WKT):

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

- from well-known binary (WKB):

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

5.2 Accesarea geometriei

First, you should find out geometry type, `wkbType()` method is the one to use — it returns a value from `Qgis.WkbType` enumeration:

```
>>> 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 enumerării `QGis.GeometryType`. Există, de asemenea, o funcie de ajutor `isMultiPart()` pentru a afla dacă o geometrie este multipart sau nu.

To extract information from geometry there are accessor functions for every vector type. How to use accessors:

```
>>> 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 multiparte există funcii accessor similare: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

5.3 Predicate i operaiuni geometrice

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

Here you have a small example that combines iterating over the features in a given layer and performing some geometric computations based on their geometries.

```
#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.

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

Proiecii suportate

6.1 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:

- specify CRS by its ID:

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

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

- `PostgisCrsId` - IDs used within PostGIS databases.
- `InternalCrsId` - IDs internally used in QGIS database.
- `EpsgCrsId` - IDs assigned by the EPSG organization

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

- specify CRS by its 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)
```

- create invalid CRS and then use one of the `create*()` functions to initialize it. In following example we use Proj4 string to initialize the projection:

```
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`.

Note that for initialization of spatial reference systems QGIS needs to lookup appropriate values in its internal database `srs.db`. Thus in case you create an independent application you need to set paths correctly with `QgsApplication.setPrefixPath()` otherwise it will fail to find the database. If you are running the commands from QGIS python console or developing a plugin you do not care: everything is already set up for you.

Accessing spatial reference system information:

```
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 QGis::units enum)
print "Map units:", crs.mapUnits()
```

6.2 Proiecii

You can do transformation between different spatial reference systems by using `QgsCoordinateTransform` class. The easiest way to use it is to create source and destination CRS and construct `QgsCoordinateTransform` instance with them. Then just repeatedly call `transform()` function to do the transformation. By default it does forward transformation, but it is capable to do also inverse transformation:

```
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.

Map canvas is implemented as `QgsMapCanvas` class in `qgis.gui` module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please make sure to read the [overview of the framework](#).

Ori de câte ori harta a fost deplasată, mărită/micorată (sau alte acțiuni care declanează 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 a afișa 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

7.1 Încapsularea suportului de hartă

Map canvas is a widget like any other Qt widget, so using it is as simple as creating and showing it:

```
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.

By default, map canvas has black background and does not use anti-aliasing. To set white background and enable anti-aliasing for smooth rendering:

```
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.)

Now it is time to add some map layers. We will first open a layer and add it to the map layer registry. Then we will set the canvas extent and set the list of layers for canvas:

```
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.

7.2 Folosirea instrumentelor în suportul de hartă

Următorul exemplu construiește o fereastră care conține un suport de hartă și instrumente de bază pentru panoramare și mărirea hărții. 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 asigurate 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 folosind 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)

```

You can put the above code to a file, e.g. `mywnd.py` and try it out in Python console within QGIS. This code will put the currently selected layer into newly created canvas:

```

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

```

Doar asigurați-vă că fișierul `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.

7.3 Benzile elastice și marcajele nodurilor

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

To show a polyline:

```

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

```

To show a polygon:

```

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

```

Rețineți 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 (opțional) inelele corespund găurilor din poligon.

Rubber bands allow some customization, namely to change their color and line width:

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

The canvas items are bound to the canvas scene. To temporarily hide them (and show again, use the `hide()` and `show()` combo. To completely remove the item, you have to remove it from the scene of the canvas:

```
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 deținut de suport)

Rubber band can be also used for drawing points, however `QgsVertexMarker` class is better suited for this (`QgsRubberBand` would only draw a rectangle around the desired point). How to use the vertex marker:

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

This will draw a red cross on position [0,0]. It is possible to customize the icon type, size, color and pen width:

```
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 și pentru eliminarea lor de pe suport, același lucru este valabil și pentru benzile elastice.

7.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă

You can write your custom tools, to implement a custom behaviour to actions performed by users on the canvas.

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

Here is an example of a map tool that allows to define a rectangular extent by clicking and dragging on the canvas. When the rectangle is defined, it prints its boundary coordinates in the console. It uses the rubber band elements described before to show the selected rectangle as it is being defined.

```
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.xMin(), r.yMin(), r.xMax(), r.yMax()

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()"))

```

7.5 Dezvoltarea elementelor personalizate pentru suportul de hartă

TODO: how to create a map canvas item

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`.

8.1 Randarea simplă

Render some layers using `QgsMapRenderer` - create destination paint device (`QImage`, `QPainter` etc.), set up layer set, extent, output size and do the rendering:

```
# 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")
```

8.2 Generarea folosind Compozitorul de hări

Compozitorul de hări reprezintă un instrument foarte util în cazul în care dorii să elaborai 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.

The central class of the composer is `QgsComposition` which is derived from `QGraphicsScene`. Let us create one:

```
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:

- `map` — this item tells the libraries where to put the map itself. Here we create a map and stretch it over the whole paper size:

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

- `label` — allows displaying labels. It is possible to modify its font, color, alignment and margin:

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

- `legendă`

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

- `bara scării`

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

- săgeată
- imagine
- formă
- tabelă

By default the newly created composer items have zero position (top left corner of the page) and zero size. The position and size are always measured in millimeters:

```
# 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)
```

A frame is drawn around each item by default. How to remove the frame:

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

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

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

The default output settings for composition are page size A4 and resolution 300 DPI. You can change them if necessary. The paper size is specified in millimeters:

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

8.2.1 Ieșire ca imagine raster

The following code fragment shows how to render a composition to a raster image:

```
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")
```

8.2.2 Ieșire în format PDF

The following code fragment renders a composition to a PDF file:

```
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 are un oarecare suport pentru parsarea expresiilor, cum ar fi SQL. Doar un mic subset al sintaxei SQL este suportat. Expresiile pot fi evaluate fie ca predicate booleene (returnând Adevărat sau Fals) sau ca funcții (care întorc o valoare scalară).

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
- funcții de conversie: 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

9.1 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')
```

9.2 Evaluarea expresiilor

9.2.1 Expresii de bază

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

9.2.2 Expresii cu entități

The following example will evaluate the given expression against a feature. “Column” is the name of the field in the layer.

```
>>> 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
```

9.2.3 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
```

9.3 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:

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of `QSettings` class. By default, this class stores settings in system’s “native” way of storing settings, that is — registry (on Windows), .plist file (on Mac OS X) or .ini file (on Unix). The [QSettings documentation](#) is comprehensive, so we will provide just a simple example:

```
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.

- **project settings** — vary between different projects and therefore they are connected with a project file. Map canvas background color or destination coordinate reference system (CRS) are examples — white background and WGS84 might be suitable for one project, while yellow background and UTM projection are better for another one. An example of usage follows:

```
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.

- **map layer settings** — these settings are related to a particular instance of a map layer with a project. They are *not* connected with underlying data source of a layer, so if you create two map layer instances of one shapefile, they will not share the settings. The settings are stored in project file, so if the user opens the project again, the layer-related settings will be there again. This functionality has been added in QGIS v1.4. The API is similar to `QSettings` — it takes and returns `QVariant` instances:

```
# 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.

11.1 Showing messages. The `QgsMessageBar` class.

Using messages boxes can be a bad idea from a user experience point of view. For showing a small info line or a warning/error messages, the QGIS message bar is usually a better option

Using the reference to the QGIS interface object, you can show a message in the message bar with the following code.

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

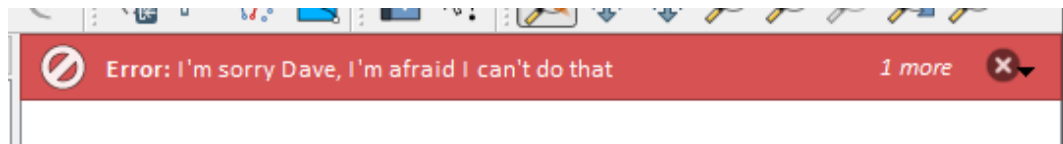


Figura 11.1: Bara de mesaje a QGIS

You can set a duration to show it for a limited time.

```
iface.messageBar().pushMessage("Error", "'Oops, the plugin is not working as \
it should", level=QgsMessageBar.CRITICAL, duration=3)
```

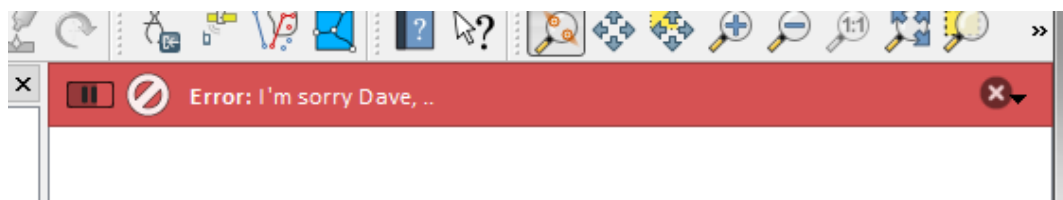


Figura 11.2: Bara de mesaje a QGIS, cu cronometru

The examples above show an error bar, but the `level` parameter can be used to creating warning messages or info messages, using the `QgsMessageBar.WARNING` and `QgsMessageBar.INFO` constants respectively.

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

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



Figura 11.3: QGIS Message bar (warning)

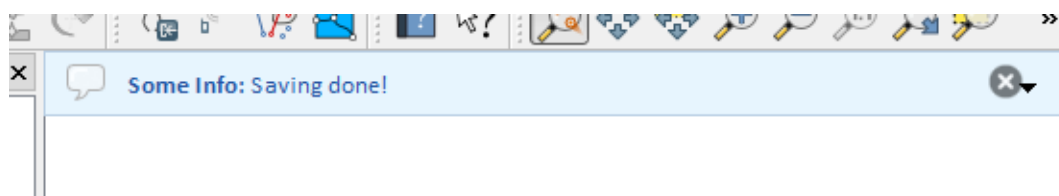


Figura 11.4: Bara de mesaje a QGIS (info)

```

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

```

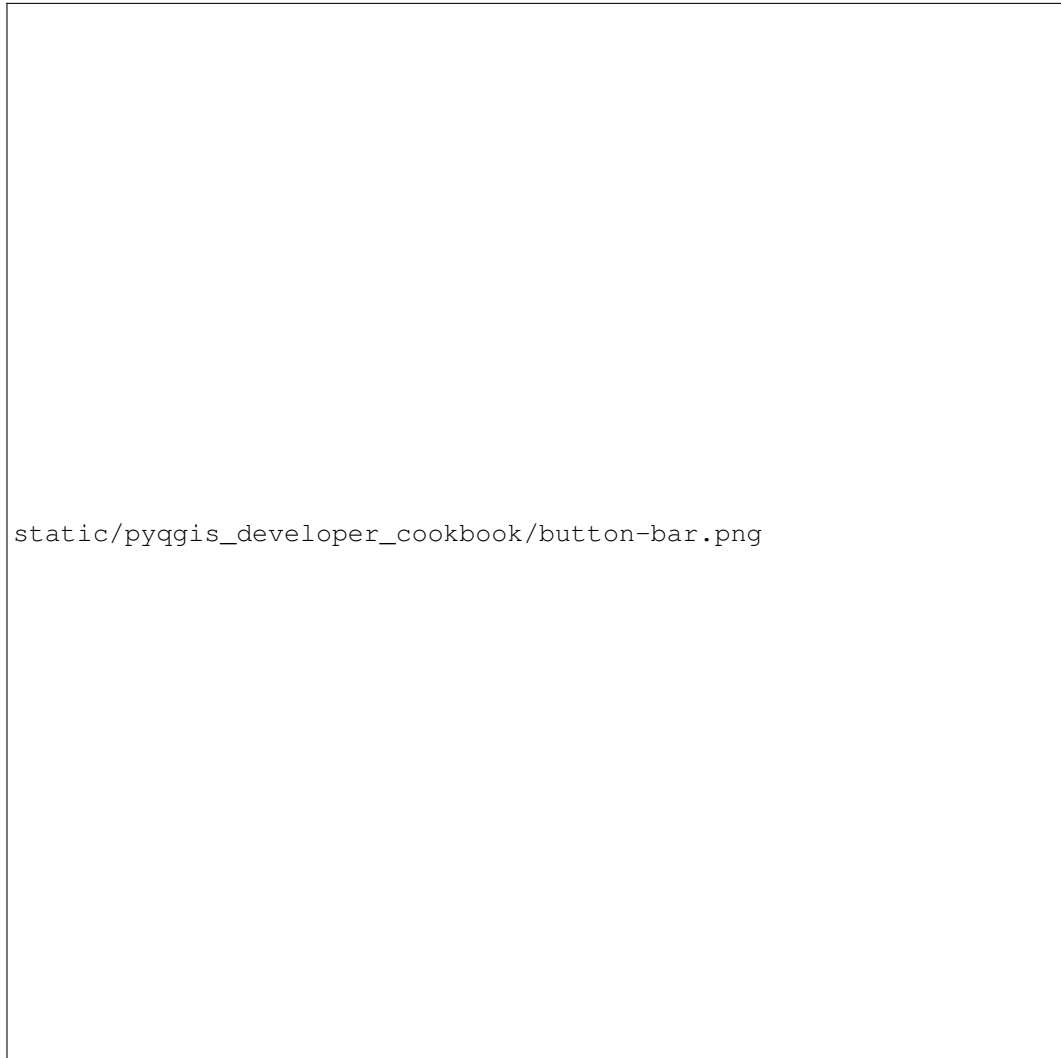


Figura 11.5: Bara de mesajă a QGIS, cu un buton

You can even use a message bar in your own dialog so you don't have to show a message box, or if it doesn't make sense to show it in the main QGIS window.

```

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)
```

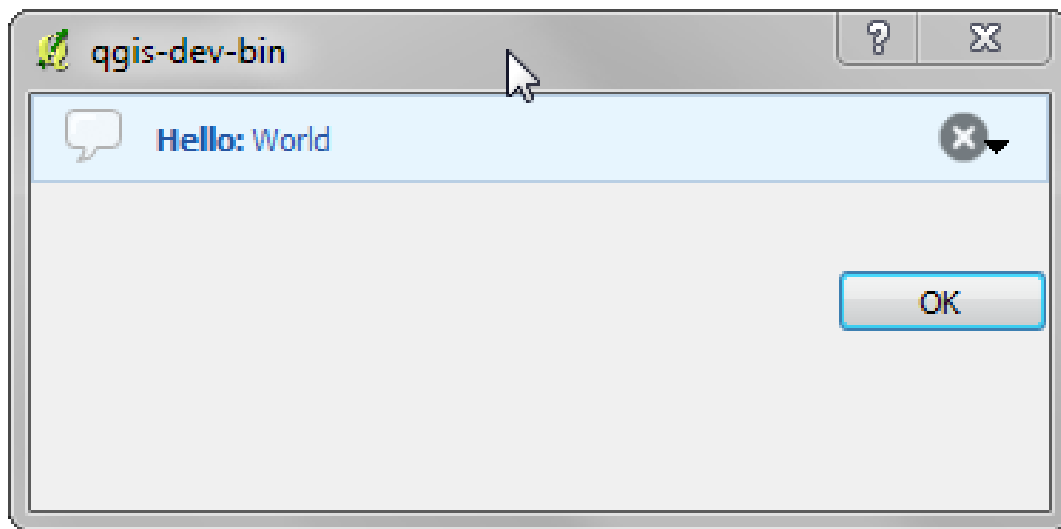


Figura 11.6: Bara de mesaje a QGIS, într-o fereastră de dialog

11.2 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)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

Also, you can use the built-in status bar to report progress, as in the next example.

```
:: 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()
```

11.3 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.

```
QgsMessageLog.logMessage("Your plugin code has been executed correctly", \
    QgsMessageLog.INFO)
QgsMessageLog.logMessage("Your plugin code might have some problems", \
```

```
QgsMessageLog.WARNING)
QgsMessageLog.logMessage("Your plugin code has crashed!", \
    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.

Python plugins are listed together with C++ plugins in QGIS plugin manager. They're being searched for in these paths:

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

Home directory (denoted by above `~`) on Windows is usually something like `C:\Documents and Settings\ (user)` (on Windows XP or earlier) or `C:\Users\ (user)`. Since Quantum GIS is using Python 2.7, subdirectories of these paths have to contain an `__init__.py` file to be considered Python packages that can be imported as plugins.

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. *Create files*: Create the files described next. A starting point (`__init__.py`). Fill in the *Metadatele plugin-ului* (`metadata.txt`) A main python plugin body (`mainplugin.py`). A form in QT-Designer (`form.ui`), with its `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. *Publish*: Publish your plugin in QGIS repository or make your own repository as an “arsenal” of personal “GIS weapons”

12.1 Scrierea unui plugin

Since the introduction of python plugins in QGIS, a number of plugins have appeared - on [Plugin Repositories wiki page](#) you can find some of them, you can use their source to learn more about programming with PyQGIS or find out whether you are not duplicating development effort. QGIS team also maintains an *Depozitul oficial al plugin-urilor python*. Ready to create a plugin but no idea what to do? [Python Plugin Ideas wiki page](#) lists wishes from the community!

12.1.1 Fiierle Plugin-ului

Here's the directory structure of our example 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 semnificatia fiierelor:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory` method and may have any other initialisation code.
- `mainPlugin.py` = Principalul codul lucrativ al plugin-ului. Conine toate informaiile cu privire la aciunile plugin-ului i ale codului principal.
- `resources.qrc` = The .xml document created by QT-Designer. Contains relative paths to resources of the forms.
- `resources.py` = Traducerea fiierului .qrc descris mai sus.
- `form.ui` = The GUI created by QT-Designer.
- `form.py` = Traducerea form.ui descris mai sus.
- `metadata.txt` = Necesari pentru QGIS >= 1.8.0. Conine informaii generale, versiunea, numele i alte metadata utilizate de ctre site-ul de plugin-uri i de ctre infrastructura plugin-ului. Începând cu QGIS 2.0 metadatale din `__init__.py` nu mai sunt acceptate, iar `metadata.txt` este necesar.

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 i nu are nevoie de conexiune la internet. Aceasta este opiunea recomandată, atât timp cât produce surse compatibile 2.0.

Warning: If you plan to upload the plugin to the *Depozitul oficial al plugin-urilor python* you must check that your plugin follows some additional rules, required for plugin *Validare*

12.2 Coninutul Plugin-ului

Aici putei găsi informaii i exemple despre ceea ce să adăugai în fiecare dintre fierele din structura de fiere descrisă mai sus.

12.2.1 | Metadatale plugin-ului

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

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

Numele metadatei	Obligatoriu	Note
nume	True	un ir scurt coninând numele pluginului
qgisMinimumVersion	True	notaie cu punct a versiunii minime QGIS
qgisMaximumVersion	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	e-mail-ul autorului, <i>nu</i> va fi afiat pe site-ul web
jurnalul schimbărilor	False	ir, poate fi pe mai multe linii, HTML nefiind permis
experimental	False	semnalizator boolean, <i>True</i> sau <i>False</i>
învechit	False	semnalizator boolean, <i>True</i> sau <i>False</i> , se aplică întregului plugin i nu doar la versiunea încărcată
etichete	False	comma separated list, spaces are allowe inside individual tags
pagina de casă	False	o adresă URL validă indicând spre pagina plugin-ului dvs.
depozit	False	o adresă URL validă pentru depozitul de cod sursă
tracker	False	o adresă validă pentru bilete i rapoartare de erori
pictogramă	False	un nume de fier sau o cale relativă (relativă la directorul de bază al pachetului comprimat al plugin-ului)
categorie	False	una din valorile <i>Raster</i> , <i>Vector</i> , <i>Bază de date</i> i <i>Web</i>

By default, plugins are placed in the *Plugins* menu (we will see in the next section how to add a menu entry for your plugin) but they can also be placed the into *Raster*, *Vector*, *Database* and *Web* menus.

A corresponding “category” metadata entry exists to specify that, so the plugin can be classified accordingly. This metadata entry is used as tip for users and tells them where (in which menu) the plugin can be found. Allowed values for “category” are: *Vector*, *Raster*, *Database* or *Web*. For example, if your plugin will be available from *Raster* menu, add this to `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*.

An example for this `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
```

12.2.2 `__init__.py`

This file is required by Python's import system. Also, Quantum GIS requires that this file contains a `classFactory()` function, which is called when the plugin gets loaded to QGIS. It receives reference to instance of `QgisInterface` and must return instance of your plugin's class from the `mainplugin.py` - in our case it's called `TestPlugin` (see below). This is how `__init__.py` should look like:

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

## any other initialisation needed
```

12.2.3 `mainPlugin.py`

This is where the magic happens and this is how magic looks like: (e.g. `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!"

```

The only plugin functions that must exist in the main plugin source file (e.g. mainPlugin.py) are: - `__init__` -> which gives access to Quantum GIS' interface - `initGui()` -> called when the plugin is loaded - `unload()` -> called when the plugin is unloaded

You can see that in the above example, the `addPluginToMenu` (<http://qgis.org/api/classQgisInterface.html#ad1af604ed>) is used. This will add the corresponding menu action to the *Plugins* menu. Alternative methods exist to add the action to a different menu. Here is a list of those methods:

- `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, aa 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 uitați să setați `QAction` și `QMenu` `objectName` unui nume specific plugin-ului dvs, astfel încât acesta să poată fi personalizat.

12.2.4 Fier de resurse

You can see that in `initGui()` we've used an icon from the resource file (called `resources.qrc` in our case):

```
<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
```

And that's all... nothing complicated :) If you've done everything correctly you should be able to find and load your plugin in the plugin manager and see a message in console when toolbar icon or appropriate menu item is selected.

Când lucrați la un plug-in real, este înțelept să stocați plugin-ul într-un alt director (de lucru), și să creați un fișier `make` care va genera UI + fișierele de resurse și să instalați plugin-ul în instalarea QGIS.

12.3 Documentație

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

Funcția `showPluginHelp()` caută fișierele de ajutor în același director ca și modulul care îl apelează. Acesta va căuta, la rândul său, în `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` și `index.html`, aflând ceea ce găsete mai întâi. Aici `ll_cc` reprezintă limba în care se afiează QGIS. Acest lucru permite multiplelor traduceri ale documentelor să fie incluse în plugin.

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

Setările IDE pentru scrierea și depanarea de plugin-uri

Although each programmer has his preferred IDE/Text editor, here are some recommendations for setting up popular IDE's for writing and debugging QGIS Python plugins.

13.1 O notă privind configurarea IDE-ului în Windows

În Linux nu este necesară nici o configurare suplimentară pentru dezvoltarea plug-in-urilor. Dar în Windows trebuie să vă asigurați că aveți aceleași setări de mediu și folosiți aceleași bibliotecile și interpretorul ca și QGIS. Cel mai rapid mod de a face acest lucru, este de a modifica fișierul batch de pornire de a QGIS.

Dacă ai folosit programul de instalare OSGeo4W, îl poți găsi în folderul bin al propriei instalări OSGeoW. Căutați ceva de genul `C:\OSGeo4W\bin\qgis-unstable.bat`.

For using [Pyscripter IDE](#), here's what you have to do:

- Faceți o copie a `qgis-unstable.bat` și redenumiți-o `pyscripter.bat`.
- Open it in an editor. And remove the last line, the one that starts `qgis`.
- Add a line that points to 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)
- Also add the argument that points to the folder where `pyscripter` can find the python dll used by `qgis`, you can find this under the bin folder of your OSGeoW install:

```
@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
```

Now when you double click this batch file it will start `pyscripter`, with the correct path.

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`.

Pentru a crea fișierul batch, urmați acești pași.

- Locate the folder where `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`
- Locate your `eclipse.exe` executable.
- Create the following script and use this to start `eclipse` when developing QGIS plugins.

```
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
```

13.2 Depanare cu ajutorul Eclipse i PyDev

13.2.1 Instalare

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

- Eclipse
- Aptana Eclipse Plugin sau PyDev
- QGIS 2.0

13.2.2 Pregătirea QGIS

E necesară efectuarea anumitor acțiuni pregătitoare pentru însui QGIS. Două plugin-uri sunt de interes: *Remote Debug* i *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, căutai *Plugin Reloader* i instalai-l. Acest lucru vă va permite să reîncărcați un plug-in, fără a fi necesare închiderea i repornirea QGIS.

13.2.3 Configurarea Eclipse

In Eclipse, create a new project. You can select *General Project* and link your real sources later on, so it does not really matter where you place this project.

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

Now in the view *Project Explorer*, your source tree pops up and you can start working with the code. You already have syntax highlighting and all the other powerful IDE tools available.

13.2.4 Configurarea depanatorului

To get the debugger working, switch to the Debug perspective in eclipse (*Window=>Open Perspective=>Other=>Debug*).

Now start the PyDev debug server by choosing *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 .

Acum puteți seta un punct de întrerupere i de îndată ce codul îl va atinge, execuția se va opri, după care veți putea inspecta starea actuală a plug-in-ului. (Punctul de întrerupere este punctul verde din imaginea de mai jos, i se poate seta printr-un dublu clic în spațiul alb din stânga liniei în care doriți un punct de întrerupere)

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.

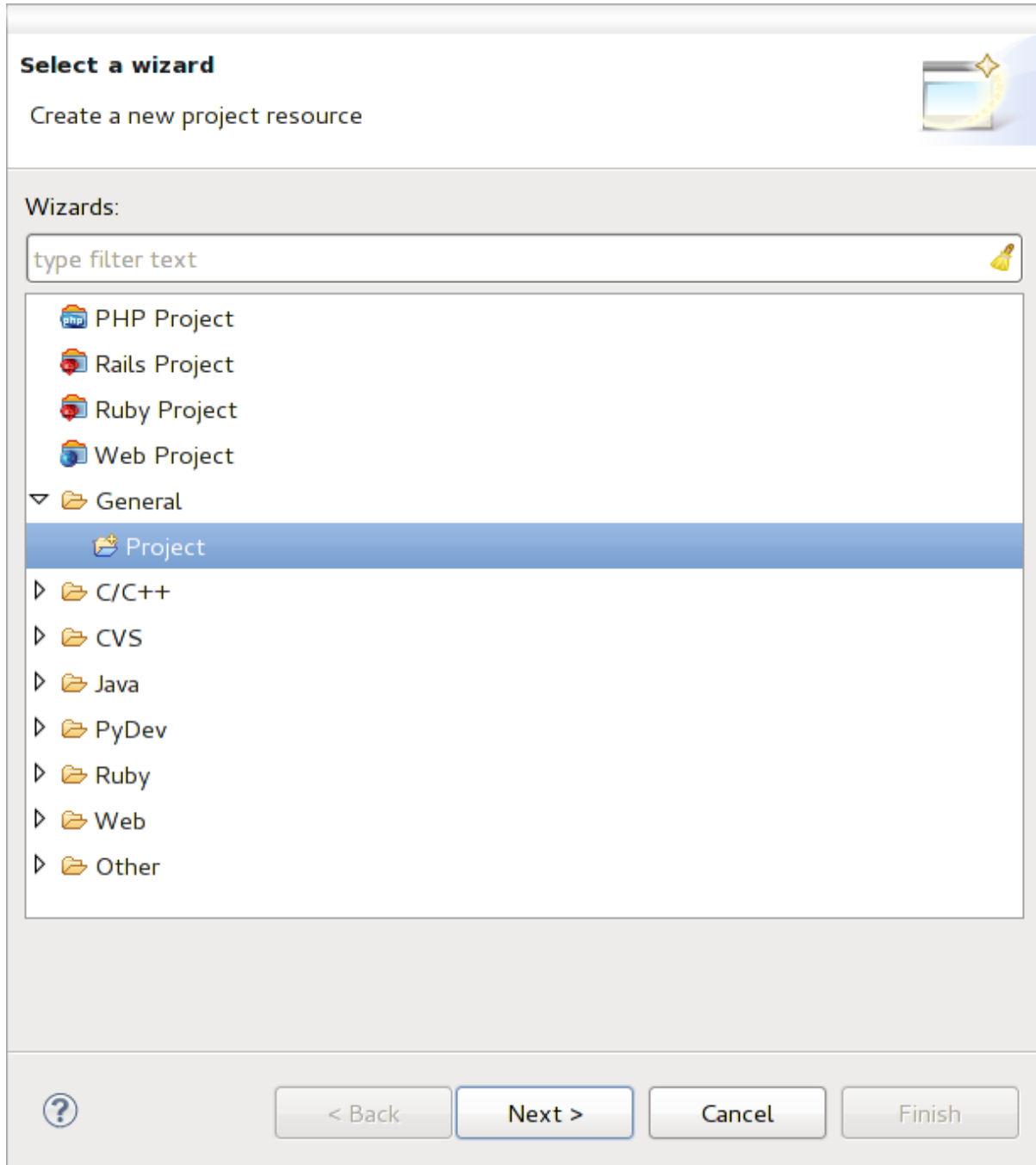


Figura 13.1: Proiectul Eclipse

```

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

```

Figura 13.2: Punct de întrerupere

Open the Console view (*Window => Show view*). It will show the Debug Server console which is not very interesting. But there is a button *Open Console* which lets you change to a more interesting PyDev Debug Console. Click the arrow next to the Open Console button and choose *PyDev Console*. A window opens up to ask you which console you want to start. Choose *PyDev Debug Console*. In case its greyed out and tells you to Start the debugger and select the valid frame, make sure that you've got the remote debugger attached and are currently on a breakpoint.

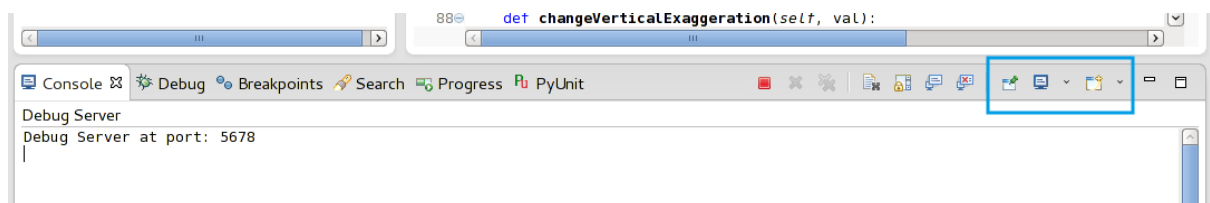


Figura 13.3: Consola de depanare PyDev

Acum avei 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.

A little bit annoying is, that everytime you enter a command, the console switches back to the Debug Server. To stop this behavior, you can click the *Pin Console* button when on the Debug Server page and it should remember this decision at least for the current debug session.

13.2.5 Configurai Eclipse pentru a înțelege API-ul

O caracteristică facilă este de a pregăti Eclipse pentru API-ul QGIS. Aceasta va permite verificarea eventualelor greeli de ortografie din cadrul codului. Dar nu doar atât, va 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.

Click *Window=>Preferences=>PyDev=>Interpreter - Python*.

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

First open the Libraries tab. Add a New Folder and choose the python folder of your QGIS installation. If you do not know where this folder is (it's not the plugins folder) open QGIS, start a python console and simply enter `qgis` and press enter. It will show you which qgis module it uses and its path. Strip the trailing `/qgis/__init__.pyc` from this path and you've got the path you are looking for.

You should also add your plugins folder here (on linux its `~/qgis/python/plugins`).

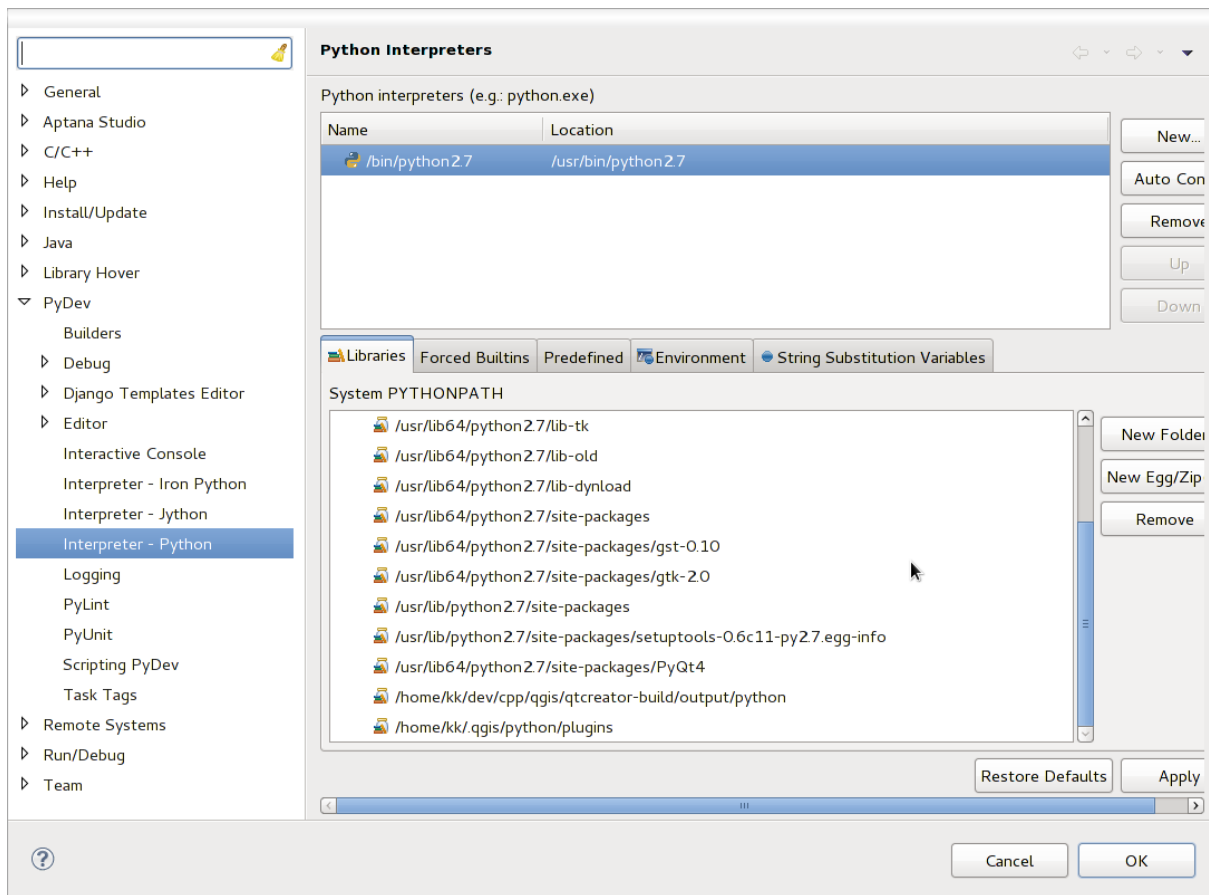


Figura 13.4: Consola de depanare PyDev

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: everytime 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](#)

13.3 Depanarea cu ajutorul PDB

If you do not use an IDE such as Eclipse, you can debug using PDB, following this steps.

First add this code in the spot where you would like to debug:

```
# 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.

În Linux rulai:

```
$ ./Qgis
```

În Mac OS X rulai:

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

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

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 cel mai bun mod de a implementa acest lucru.

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

14.1 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
```

Methods for reading and writing specific information to the project file can also be added:

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

When loading a project containing such a layer, a factory class is needed:

```
class WatermarkPluginLayerType(QgsPluginLayerType):
    def __init__(self):
        QgsPluginLayerType.__init__(self, WatermarkPluginLayer.LAYER_TYPE)

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

You can also add code for displaying custom information in the layer properties:

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

Compatibilitatea cu versiunile QGIS anterioare

15.1 Meniul plugin-ului

If you place your plugin menu entries into one of the new menus (*Raster*, *Vector*, *Database* or *Web*), you should modify the code of the `initGui()` and `unload()` functions. Since these new menus are available only in QGIS 2.0, the first step is to check that the running QGIS version has all necessary functions. If the new menus are available, we will place our plugin under this menu, otherwise we will use the old *Plugins* menu. Here is an example for *Raster* menu:

```
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 i 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 putei găsi instrucțiuni de împachetare i de pregătire a plugin-ului, pentru a lucra bine cu programul de instalare. Sau, în cazul în care ai dori să înfiinai un depozit propriu pentru plugin-uri, creai un simplu fiier XML, care va lista plugin-urile i metadatele lor, exemplu pe care îl putei vedea în *depozite pentru plugin-uri*.

16.1 Depozitul oficial al plugin-urilor python

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

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

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

16.1.1 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 alii, atât timp cât acetia sunt prezeni în lista *proprietarilor* de plugin-uri
- un anumit plug-in pot fi ters i editat doar de utilizatorii *staff-ului* i de către *proprietarii* plugin-uri
- în cazul în care un utilizator fără permisiunea *plugins.can_approve* încarcă o nouă versiune, versiunea plug-inului nu va fi aprobată, din start.

16.1.2 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 crete încrederea în creatorul sau *proprietarul* plugin-ului.

16.1.3 Validare

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

Iată câteva reguli de validare pe care ar trebui să le cunoatei atunci când dorii să încarci un plugin în depozitul oficial:

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

16.1.4 Structura plugin-ului

Following the validation rules the compressed (.zip) package of your plugin must have a specific structure to validate as a functional plugin. As the plugin will be unzipped inside the users plugins folder it must have it's own directory inside the .zip file to not interfere with other plugins. Mandatory files are: `netadata.txt` and `__init__.py` But it would be nice to have a `README.py` and of course an icon to represent the plugin (`resources.qrc`). Following is an example of how a `plugin.zip` should look like.

```
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.md
|-- 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.

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

In the plug-in add to the `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)
```

To `unload()` add:

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

The method that is called when F7 is pressed:

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

17.2 How to toggle Layers (work around)

As there is currently no method to directly access the layers in the legend, here is a workaround how to toggle the layers using layer transparency:

```
def toggleLayer(self, lyrNr):
    lyr = self.iface.mapCanvas().layer(lyrNr)
    if lyr:
        cTran = lyr.getTransparency()
        lyr.setTransparency(0 if cTran > 100 else 255)
        self.iface.mapCanvas().refresh()
```

The method requires the layer number (0 being the top most) and can be called by:

```
self.toggleLayer(3)
```

17.3 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 from current layer")
else:
    QMessageBox.critical(self.iface.mainWindow(),"Error","Please select a layer")
```

The method requires one parameter (the new value for the attribute field of the selected feature(s)) and can be called by:

```
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 a QGIS. Biblioteca:

- creează graful matematic din datele geografice (straturi vectoriale de tip polilinie)
- implementeaza metoda de bază a teoriei grafurilor (actualmente doar algoritmul lui Dijkstra)

Network analysis library was created by exporting basic functions from RoadGraph core plugin and now you can use its methods in plugins or directly from Python console.

18.1 Informații generale

Briefly typical use case can be described as:

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)

18.2 Building graph

The first thing you need to do — is to prepare input data, that is to convert vector layer into graph. All further actions will use this graph, not the layer.

As a source we can use any polyline vector layer. Nodes of the polylines become graph vertices, and segments of the polylines are graph edges. If several nodes have the same coordinates then they are the same graph vertex. So two lines that have a common node become connected to each other.

Additionally, during graph creation it is possible to “fix” (“tie”) to the input vector layer any number of additional points. For each additional point a match will be found — closest graph vertex or closest graph edge. In the latter case the edge will be splitted and new vertex added.

As the properties of the edge a vector layer attributes can be used and length of the edge.

Converter from vector layer to graph is developed using **Builder** programming pattern. For graph construction response so-called **Director**. There is only one Director for now: `QgsLineVectorLayerDirector`. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create graph. Currently, as in the case with the director, only one builder exists: `QgsGraphBuilder`, that creates `QgsGraph` objects. You may want to implement your own builders that will build a graphs compatible with such libraries as `BGL` or `NetworkX`.

To calculate edge properties programming pattern **strategy** is used. For now only `QgsDistanceArcProperter` strategy is available, that takes into account the length of the route. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses strategy that compute travel time using edge length and speed value from attributes.

It's time to dive in the process.

First of all, to use this library we should import networkanalysis module:

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

Then create 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 roads direction.
# unilateral roads with direct direction have attribute value "yes",
# unilateral roads with reverse direction - "1", and accordingly bilateral
# roads - "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 )
```

To construct a director we should pass vector layer, that will be used as source for graph and information about allowed movement on each road segment (unilateral or bilateral movement, direct or reverse direction). Here is full list of this parameters:

- vl — vector layer used to build graph
- directionFieldId — index of the attribute table field, where information about roads directions is stored. If -1, then don't use this info at all
- directDirectionValue — field value for roads with direct direction (moving from first line point to last one)
- reverseDirectionValue — field value for roads with reverse direction (moving from last line point to first one)
- bothDirectionValue — field value for bilateral roads (for such roads we can move from first point to last and from last to first)
- defaultDirection — default road direction. This value will be used for those roads where field directionFieldId is not set or have some value different from above.

It is necessary then to create strategy for calculating edge properties:

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy:

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

Now we can create builder, which will create graph. QgsGraphBuilder constructor takes several arguments:

- crs — sistemul de coordonate de referință de utilizat. Argument obligatoriu.
- offtEnabled — utilizai sau nu reproiectarea “din zbor”. În mod implicit const:True (folosii OTF).
- topologyTolerance — tolerana 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 )
```

Also we can set several points, which will be used in analysis. For example:

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

Now all is in place so we can build graph and “tie” points to it:

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

Building graph can take some time (depends on number of features in a layer and layer size). `tiedPoints` is a list with coordinates of “tied” points. When build operation is finished we can get graph and use it for the analysis:

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

With the next code we can get indexes of our points:

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

18.3 Analiza grafului

Networks analysis is used to find answers on two questions: which vertices are connected and how to find a shortest path. To solve this problems network analysis library provides Dijkstra’s algorithm.

Dijkstra’s algorithm finds the best route from one of the vertices of the graph to all the others and the values of the optimization parameters. The results can be represented as shortest path tree.

The shortest path tree is as oriented weighted graph (or more precisely — tree) with the following properties:

- only one vertex have no incoming edges — the root of the tree
- all other vertices have only one incoming edge
- if vertex B is reachable from vertex A, then path from A to B is single available path and it is optimal (shortest) on this graph

To get shortest path tree use methods `shortestTree()` and `dijkstra()` of `QgsGraphAnalyzer` class. It is recommended to use method `dijkstra()` because it works faster and uses memory more efficiently.

The `shortestTree()` method is useful when you want to walk around the shortest path tree. It always creates new graph object (`QgsGraph`) and accepts three variables:

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

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

Metoda `dijkstra()` are aceleai argumente, dar întoarce două tablouri. În prima matrice, elementul `i` conine indexul marginii de intrare, sau -1 în cazul în care nu există margini de intrare. În a doua matrice, elementul `i` conine distanța de la rădăcina arborelui la nodul `i`, sau `DOUBLE_MAX` dacă rădăcina nodului este imposibil de căutat.

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

Here is very simple code to display shortest path tree using graph created with `shortestTree()` method (select linestring layer in TOC and replace coordinates with yours one). **Warning:** use this code only as an example, it creates a lots of `QgsRubberBand` objects and may be slow on large datasets.

```
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() )
```

18.3.1 Finding shortest path

To find optimal path between two points the following approach is used. Both points (start A and end B) are “tied” to graph when it builds. Than using methods `shortestTree()` or `dijkstra()` we build shortest tree with root in the start point A. In the same tree we also found end point B and start to walk through tree from point B to point A. Whole algorithm can be written as:

```
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

```

At this point we have path, in the form of the inverted list of vertices (vertices are listed in reversed order from end point to start one) that will be visited during traveling by this path.

Here is the sample code for QGIS Python Console (you will need to select linestring layer in TOC and replace coordinates in the code with yours) that uses method `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)

```

And here is the same sample but using `dijkstra()` method:

```
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)
```

18.3.2 Areas of the availability

Area of availability for vertex A is a subset of graph vertices, that are accessible from vertex A and cost of the path from A to this vertices are not greater than some value.

More clearly this can be shown with the following example: “There is a fire station. What part of city fire command can reach in 5 minutes? 10 minutes? 15 minutes?”. Answers on these questions are fire station’s areas of availability.

To find areas of availability we can use method `dijkstra()` of the `QgsGraphAnalyzer` class. It is enough to compare elements of cost array with predefined value. If `cost[i]` is less or equal than predefined value, then vertex `i` is inside area of availability, otherwise — outside.

More difficult it is to get borders of area of availability. Bottom border — is a set of vertices that are still accessible, and top border — is a set of vertices which are not accessible. In fact this is simple: availability border passed on such edges of the shortest path tree for which start vertex is accessible and end vertex is not accessible.

Here is an example:

```

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
 - Fișiere GPX, 6
 - Geometrii MySQL, 6
 - straturi cu text delimitat, 5
 - Straturi OGR, 5
 - Straturi PostGIS, 5
 - straturi raster, 6
 - Straturi SpatiaLite, 6
 - straturi vectoriale, 5
 - Straturi WMS, 7
 - aisteme de coordonate de referință, 29
 - API, 1
 - aplicații personalizate
 - în execuție, 3
 - Python, 2
 - calcularea valorilor, 40
 - consolă
 - Python, 1
 - entități
 - straturi vectoriale iterarea, 13
 - expresii, 40
 - evaluare, 42
 - parsare, 41
 - Fișiere GPX
 - încărcare, 6
 - filtrare, 40
 - furnizor de memorie, 18
 - geometrie
 - accesare, 27
 - construire, 27
 - manipulare, 25
 - predicată și operațiuni, 28
 - Geometrii MySQL
 - încărcare, 6
 - ieire
 - folosirea Compozitorului de Hărți, 37
 - imagine raster, 39
 - PDF, 39
 - index spațial
 - folosind, 16
 - interogare
 - straturi raster, 11
 - iterarea
 - entități, straturi vectoriale, 13
 - metadata, 56
 - metadata.txt, 56
 - personalizat
 - rendere, 23
 - plugin-uri, 69
 - apelarea unei metode printr-o combinație rapidă de taste, 71
 - atributele de acces ale entităților selectate, 71
 - comutarea straturilor, 71
 - depozitul oficial al plugin-urilor python, 69
 - dezvoltare, 51
 - documentație, 58
 - fișier de resurse, 58
 - fragmente de cod, 58
 - implementare help, 58
 - lansarea, 64
 - metadata.txt, 54, 56
 - scriere, 53
 - scriere cod, 54
 - testare, 64
 - proiecții, 30
 - Python
 - aplicații personalizate, 2
 - consolă, 1
 - dezvoltarea plugin-urilor, 51
 - plugin-uri, 1
 - randare hartă, 35
 - simplic, 37
 - rastere
 - multibandă, 11
 - simplică bandă, 10
 - recitare
 - straturi raster, 11
 - registru straturilor de hartă, 7
 - adăugarea unui strat, 7
 - render cu simbol gradual, 20
 - render cu simbologie clasificată, 19
-

- render cu un singur simbol , 19
- rendere
 - personalizat, 23
- resources.qrc, 58

- setări
 - citire, 43
 - global, 45
 - proiect, 45
 - stocare, 43
 - strat de hartă, 46
- simbologie
 - render cu simbol clasificat, 19
 - render cu simbol gradual, 20
 - render cu un singur simbol , 19
 - vechi, 25
- simboluri
 - lucrul cu, 21
- straturi cu text delimitat
 - încărcare, 5
- Straturi OGR
 - încărcare, 5
- Straturi PostGIS
 - încărcare, 5
- straturi raster
 - încărcare, 6
 - detalii, 9
 - folosind, 7
 - interogare, 11
 - recitare, 11
 - stil de desenare, 9
- Straturi SpatialLite
 - încărcare, 6
- straturi vectoriale
 - încărcare, 5
 - editare, 14
 - iterarea entității, 13
 - scris, 17
 - simbologie, 19
- Straturi WMS
 - încărcare, 7
- straturile plugin-ului, 64
 - subclasarea QgsPluginLayer, 65
- straturile simbolului
 - crearea tipurilor personalizate, 21
 - lucrul cu, 21
- suportul hărții, 30
 - încapsulare, 31
 - arhitectură, 31
 - benzi de cauciuc, 33
 - dezvoltarea elementelor personalizate pentru suportul de hartă , 35
 - dezvoltarea instrumentelor de hartă personalizate, 34
 - instrumente pentru hartă, 32
 - marcaje vertex, 33

- tipărire hartă, 35