
PyQGIS developer cookbook

Release 2.14

QGIS Project

August 08, 2017

1	Introducere	1
1.1	Rularea codului Python atunci când pornește QGIS	1
1.2	Consola Python	2
1.3	Plugin-uri Python	3
1.4	Aplicații Python	3
2	Încărcarea proiectelor	7
3	Încărcarea Straturilor	9
3.1	Straturile Vectoriale	9
3.2	Straturile Raster	11
3.3	Registrul Straturilor de Hartă	11
4	Utilizarea straturilor raster	13
4.1	Detaliile stratului	13
4.2	Render	13
4.3	Recitirea straturilor	15
4.4	Interogarea valorilor	15
5	Utilizarea straturilor vectoriale	17
5.1	Obținerea informațiilor despre atribute	17
5.2	Selectarea entităților	18
5.3	Iterații în straturile vectoriale	18
5.4	Modificarea straturilor vectoriale	20
5.5	Modificarea straturi vectoriale prin editarea unui tampon de memorie	21
5.6	Crearea unui index spațial	23
5.7	Scrierea straturilor vectoriale	23
5.8	Furnizorul de memorie	24
5.9	Aspectul (simbologia) straturilor vectoriale	25
5.10	Lecturi suplimentare	33
6	Manipularea geometriei	35
6.1	Construirea geometriei	35
6.2	Accesarea geometriei	36
6.3	Predicate și operațiuni geometrice	36
7	Proiecții suportate	39
7.1	Sisteme de coordonate de referință	39
7.2	Proiecții	40
8	Folosirea suportului de hartă	41
8.1	Încapsularea suportului de hartă	41
8.2	Folosirea instrumentelor în suportul de hartă	42

8.3	Benzile elastice și marcajele nodurilor	43
8.4	Dezvoltarea instrumentelor personalizate pentru suportul de hartă	44
8.5	Dezvoltarea elementelor personalizate pentru suportul de hartă	45
9	Randarea hărților și imprimarea	47
9.1	Randarea simplă	47
9.2	Randarea straturilor cu diferite CRS-uri	48
9.3	Generarea folosind Compozitorul de hărți	48
10	Expresii, filtrarea și calculul valorilor	51
10.1	Parsarea expresiilor	52
10.2	Evaluarea expresiilor	52
10.3	Exemple	53
11	Citirea și stocarea setărilor	55
12	Comunicarea cu utilizatorul	57
12.1	Afișarea mesajelor. Clasa QgsMessageBar	57
12.2	Afișarea progresului	58
12.3	Jurnalizare	59
13	Dezvoltarea plugin-urilor Python	61
13.1	Scrierea unui plugin	62
13.2	Conținutul Plugin-ului	62
13.3	Documentație	67
13.4	Traducerea	67
14	Setările IDE pentru scrierea și depanarea de plugin-uri	69
14.1	O notă privind configurarea IDE-ului în Windows	69
14.2	Depanare cu ajutorul Eclipse și PyDev	70
14.3	Depanarea cu ajutorul PDB	74
15	Utilizarea straturilor plugin-ului	75
15.1	Subclasarea QgsPluginLayer	75
16	Compatibilitatea cu versiunile QGIS anterioare	77
16.1	Meniul plugin-ului	77
17	Lansarea plugin-ului dvs.	79
17.1	Metadate și nume	79
17.2	Codul și ajutorul	79
17.3	Depozitul oficial al plugin-urilor python	80
18	Fragmente de cod	83
18.1	Cum să apelăm o metodă printr-o combinație rapidă de taste	83
18.2	Inversarea Stării Straturilor	83
18.3	Cum să accesați tabelul de atribute al entităților selectate	83
19	Scrierea unui plugin Processing	85
19.1	Crearea unui plug-in care adaugă un furnizor de algoritm	85
19.2	Crearea unui plug-in care conține un set de script-uri de procesare	85
20	Biblioteca de analiză a rețelelor	87
20.1	Informații generale	87
20.2	Construirea unui graf	87
20.3	Analiza grafului	89
21	Plugin-uri Python pentru Serverul QGIS	95
21.1	Arhitectura Plugin-urilor de Filtrare de pe Server	95
21.2	Tratarea excepțiilor provenite de la un plugin	97

21.3	Scrierea unui plugin pentru server	97
21.4	Plugin-ul de control al accesului	100
	Index	105

Introducere

- Rularea codului Python atunci când pornește QGIS
 - Variabila de mediu `PYQGIS_STARTUP`
 - Fișierul `startup.py`
- Consola Python
- Plugin-uri Python
- Aplicații Python
 - Utilizarea PyQGIS în script-uri de sine stătătoare
 - Utilizarea PyQGIS în aplicații personalizate
 - Rularea Aplicațiilor Personalizate

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

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

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

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

Legăturile cu Python sunt, de asemenea, disponibile pentru QGIS Server:

- începând cu versiunea 2.8, plugin-urile Python sunt, de asemenea, disponibile pentru QGIS Server (v. Plugin-urile Serverului Python)
- începând cu versiunea 2.11 (master-ul din 2015-08-11), biblioteca Serverului QGIS are legături către Python, care pot fi utilizate pentru a încorpora QGIS Server într-o aplicație Python.

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

O metodă bună de învățare, atunci când lucrați cu plugin-uri, este de a le descărca din [depozitul de plugin-uri](#), apoi să le examinați codul. De asemenea, folderul `python/plugins/` din instalarea QGIS conține unele plugin-uri din care puteți învăța, în scopul dezvoltării unor similare, capabile să efectueze majoritatea sarcinilor comune.

1.1 Rularea codului Python atunci când pornește QGIS

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

1.1.1 Variabila de mediu PYQGIS_STARTUP

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

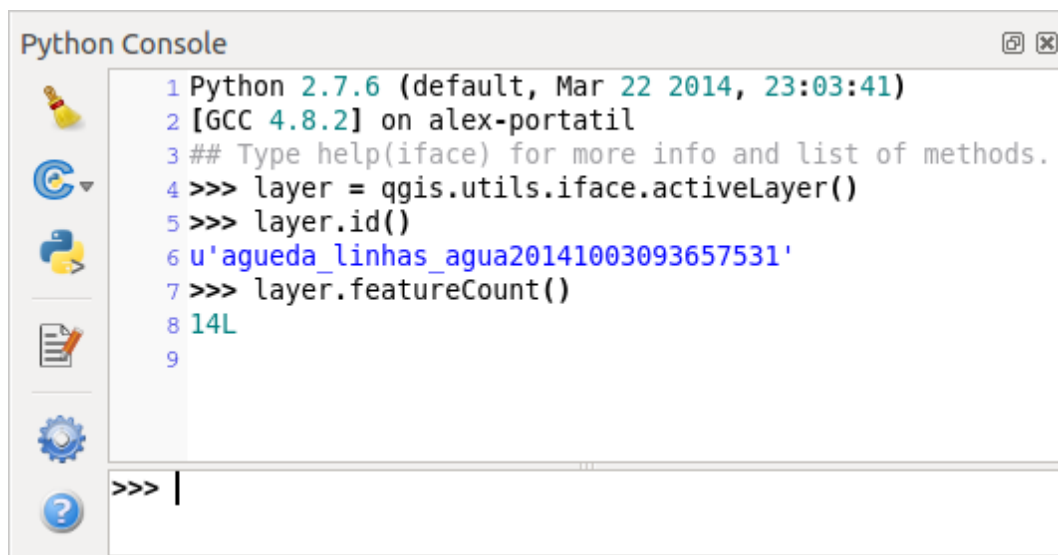
Această metodă este rar utilizată, dar merită menționată, deoarece reprezintă una din metodele de a rula cod Python în QGIS, și pentru că acest cod se va executa înainte de finalizarea inițializării QGIS. Această metodă este foarte utilă pentru curățarea `sys.path`, care poate conține căi nedorite, sau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

1.1.2 Fișierul `startup.py`

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

1.2 Consola Python

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



```

Python Console
1 Python 2.7.6 (default, Mar 22 2014, 23:03:41)
2 [GCC 4.8.2] on alex-portatil
3 ## Type help(iface) for more info and list of methods.
4 >>> layer = qgis.utils.iface.activeLayer()
5 >>> layer.id()
6 u'agueda_linhas_agua20141003093657531'
7 >>> layer.featureCount()
8 14L
9
>>> |

```

Figure 1.1: Consola Python din QGIS

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

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

```

from qgis.core import *
import qgis.utils

```

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

1.3 Plugin-uri Python

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

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

Crearea de plugin-uri în Python este simplă, instrucțiuni detaliate găsindu-se în :ref: *developing_plugins*.

Note: Plugin-urile Python sunt, de asemenea, disponibile pentru QGIS Server (*label_qgisserver*), v. *Plugin-uri Python pentru Serverul QGIS* pentru mai multe detalii.

1.4 Aplicații Python

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

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

Aplicațiile personalizate PyQGIS, sau script-urile de sine stătătoare, trebuie să fie configurate pentru a putea găsi resursele QGIS, cum ar fi informațiile despre proiecție, furnizorii pentru citirea straturilor vectoriale și raster etc. Resursele QGIS sunt inițializate prin adăugarea a câtorva rânduri la începutul aplicației sau script-ului. Codul de inițializare QGIS este similar, atât pentru aplicațiile personalizate cât și pentru script-urile de sine stătătoare, exemplele fiind prezentate mai jos.

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

1.4.1 Utilizarea PyQGIS în script-uri de sine stătătoare

Pentru a starta un script independent, inițializați resursele QGIS la începutul script-ului, similar codului următor:

```
from qgis.core import *

# supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# create a reference to the QgsApplication, setting the
# second argument to False disables the GUI
qgs = QgsApplication([], False)

# load providers
qgs.initQgis()

# Write your code here to load some layers, use processing algorithms, etc.

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

Începem prin importarea modulului `qgis.core`, urmată de configurarea prefixului căii. Prefixul căii este reprezentat de locația în care este instalat QGIS pe sistemul dumneavoastră. Configurarea sa în cadrul scriptului are loc prin apelarea metodei `setPrefixPath`. Al doilea argument al lui `setPrefixPath` este setat pe `TRUE`, care stabilește dacă sunt folosite căile implicite.

Calea de instalare a QGIS variază în funcție de platformă; cel mai simplu mod de a o identifica pe cea din sistemul dvs. este de a utiliza *Consola Python* din interiorul QGIS și analizând rezultatul generat de execuția `QgsApplication.prefixPath()`.

După configurarea căii prefixului, în variabila `qgs` vom salva o referință către `QgsApplication`. Al doilea argument este setat la `False`, ceea ce indică faptul că nu avem de gând să utilizăm un GUI, atât timp cât dorim să scriem un script de sine stătător. Având `QgsApplication` configurată, vom încărca furnizorii de date QGIS și registrul stratului, prin apelarea metodei `qgs.initQgis()`. Aplicația QGIS fiind inițializată, suntem gata să scriem restul script-ului. În cele din urmă, vom încheia printr-un apel la `qgs.exitQgis()`, pentru a elimina din memorie furnizorii de date și registrul stratului.

1.4.2 Utilizarea PyQGIS în aplicații personalizate

Singura diferență dintre *Utilizarea PyQGIS în script-uri de sine stătătoare* și o aplicație PyQGIS particularizată este dată de al doilea argument, la instanțierea `QgsApplication`. Vom transmite `True` în loc de `False`, pentru a indica faptul că intenționăm să utilizăm o interfață grafică.

```
from qgis.core import *

# supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# create a reference to the QgsApplication
# setting the second argument to True enables the GUI, which we need to do
# since this is a custom application
qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

# Write your code here to load some layers, use processing algorithms, etc.

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

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

1.4.3 Rularea Aplicațiilor Personalizate

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

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

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

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

Deși calea către modulele PyQGIS este de acum cunoscută, ele depind totuși de bibliotecile `qgis_core` și `qgis_gui` (modulele Python servesc numai pentru intermedierea apelării). Calea către aceste biblioteci este de

obicei necunoscută sistemului de operare, astfel că veți obține iarăși o eroare de import (mesajul putând varia în funcție de sistem):

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

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

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

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

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

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

Încărcarea proiectelor

Uneori trebuie să încărcați un proiect existent dintr-un plugin, sau (mai des) atunci când dezvoltați o aplicație QGIS independentă în Python (v.: *Aplicații Python*).

Pentru a încărca un proiect în aplicația QGIS curentă aveți nevoie de obiectul `QgsProject` `instance()`, căruia să-i apelați metoda `read()`, și să o transferați obiectului `QFileInfo`, care conține calea de unde va fi încărcat proiectul:

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

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

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

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

Note: Dacă scrieți o aplicație QGIS de sine stătătoare, în scopul sincronizării proiectului încărcat în canevas, trebuie să instanțiați o clasă `QgsLayerTreeMapCanvasBridge`, ca în exemplul de mai jos:

```
bridge = QgsLayerTreeMapCanvasBridge( \
    QgsProject.instance().layerTreeRoot(), canvas)
# Now you can safely load your project and see it in the canvas
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
```

Încărcarea Straturilor

- Straturile Vectoriale
- Straturile Raster
- Registrul Straturilor de Hartă

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

3.1 Straturile Vectoriale

Pentru a încărca un strat vectorial, specificați 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 șir specific pentru fiecare furnizor de date vectoriale în parte. Numele stratului se va afișa în lista straturilor. Este important să se verifice dacă stratul a fost încărcat cu succes. În cazul neîncărcării cu succes, va fi returnată o instanță de strat nevalid.

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

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

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

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

- Biblioteca OGR (fișiere shape și multe alte formate de fișiere) — sursa de date reprezintă calea către fișier:
 - pentru fișiere shape:

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

- pentru dxf (notați opțiunile interne din URI-ul sursei de date):

```
uri = "/path/to/dxffile/file.dxf|layername=entities|geometrytype=Point"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
```

- Baza de date PostGIS — sursa de date este constituită dintr-un șir cu toate informațiile necesare pentru a crea o conexiune la baza de date PostgreSQL. Clasa `QgsDataSourceURI` poate genera acest șir pentru

dvs. Rețineți că aplicația QGIS trebuie să fie compilată cu suport pentru 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", "johnny", "xxx")
# set database schema, table name, geometry column and optionally
# subset (WHERE clause)
uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

vlayer = QgsVectorLayer(uri.uri(), "layer name you like", "postgres")
```

- CSV sau alte fișiere text delimitate — pentru a deschide un fișier având punct și virgulă ca delimitator, iar câmpurile “x” și “y” ca și coordonate x, respectiv y, ar trebui să folosiți 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")
```

Notă: încă de la QGIS versiunea 1.7, șirul furnizorului este structurat ca un URL, astfel încât calea trebuie să fie prefixată cu `file://`. De asemenea, sunt permise geometrii în format WKT (well known text), ca o alternativă la câmpurile “x” și “y”, permițând și specificarea sistemului de coordonate de referință. De exemplu:

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

- Fișiere GPX — furnizorul de date “gpx” citește urme, rute și puncte de referință din fișiere gpx. Pentru a deschide un fișier, tipul (urmă/traseu/punct de referință) 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 SpatiaLite — începând cu QGIS v1.1. În mod similar bazelor de date PostGIS, QgsDataSourceURI se poate utiliza pentru generarea identificadorului 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 o reprezintă șirul 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 și cu ajutorul furnizorului WFS:

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

Identificadorul URI poate fi creat folosindu-se 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))
```


Note: Puteți schimba sursa de date a unui strat existent, prin apelarea `setDataSource()` asupra unei instanțe `QgsMarkerSymbolV2`, ca în următorul exemplu de cod:

```
# layer is a vector layer, uri is a QgsDataSourceURI instance
layer.setDataSource(uri.uri(), "layer name you like", "postgres")
```

3.2 Straturile Raster

Pentru accesarea fișierelor raster este utilizată biblioteca GDAL. Aceasta acceptă o gamă largă de formate de fișiere. În cazul în care aveți probleme cu deschiderea unor fișiere, verificați dacă GDAL are suport pentru formatul respectiv (nu toate formatele sunt disponibile în mod implicit). Pentru a încărca un raster dintr-un fișier, specificați numele fișierului și numele de bază:

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

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

```
iface.addRasterLayer("/path/to/raster/file.tif", "layer name you like")
```

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

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

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

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

Alternativ, puteți încărca un strat raster de pe un server WMS. Cu toate acestea, în prezent, nu este posibilă accesarea din API a răspunsului `GetCapabilities` — trebuie să cunoșteți straturile dorite:

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

3.3 Registrul Straturilor de Hartă

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

Adăugarea unui strat la registru:

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

Straturile sunt distruse în mod automat la ieșire; cu toate acestea, dacă doriți să ștergeți stratul în mod explicit, atunci folosiți:

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

Pentru o listă a straturilor încărcate și a id-urilor acestora, folosiți:

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

Utilizarea straturilor raster

- Detaliile stratului
- Render
 - Rastere cu o singură bandă
 - Rastere multibandă
- Recitirea straturilor
- Interogarea valorilor

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

4.1 Detaliile stratului

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

```

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

```

4.2 Render

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

Pentru a interoga renderul curent:

```
>>> rlayer.renderer()
<qgis._core.QgsSingleBandPseudoColorRenderer object at 0x7f471c1da8a0>
>>> rlayer.renderer().type()
u'singlebandpseudocolor'
```

Pentru a seta un render folosiți metoda `setRenderer()` a clasei `QgsRasterLayer`. Există mai multe clase de renderer disponibile (derivate din `QgsRasterRenderer`):

- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`
- `QgsSingleBandColorDataRenderer`
- `QgsSingleBandGrayRenderer`
- `QgsSingleBandPseudoColorRenderer`

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

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

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

4.2.1 Rastere cu o singură bandă

Să presupunem că vrem să randăm stratul raster (presupunând că are o singură bandă), folosind culori care variază de la verde la galben (pentru valori ale pixelilor de la 0 la 255). În prima etapă, vom pregăti obiectul `QgsRasterShader` și îi vom configura funcția de nuanțare:

```
>>> fcn = QgsColorRampShader()
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
          QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn.setColorRampItemList(lst)
>>> shader = QgsRasterShader()
>>> shader.setRasterShaderFunction(fcn)
```

Nuanțatorul mapează culorile hărții, așa cum este specificat în harta sa de culori. Harta de culoare reprezintă o listă de elemente cu valorile pixelilor și culoarea asociată acestora. Există trei moduri de interpolare a valorilor:

- liniar (`INTERPOLATED`): 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

În a doua etapă, vom asocia acest nuanțator cu stratul raster:

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

Numărul 1 din codul de mai sus reprezintă numărul benzii (benzile raster sunt indexate începând de la unu).

4.2.2 Rastere multibandă

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

```
rlayer.renderer().setGreenBand(1)
rlayer.renderer().setRedBand(2)
```

Atunci când este necesară doar o singură bandă pentru vizualizarea raster, poate fi aleasă desenarea unei singure benzi — cu tonuri de gri sau cu pseudoculori.

4.3 Recitirea straturilor

Dacă schimbați simbologia stratului și ați vrea să vă asigurați că schimbările sunt imediat vizibile pentru utilizator, puteți apela aceste metode

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

Primul apel garantează că imaginea din cache a stratului este ștearsă 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.

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

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

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

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

4.4 Interogarea valorilor

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

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

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

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

Utilizarea straturilor vectoriale

- Obținerea informațiilor despre atribute
- Selectarea entităților
- Iterații în straturile vectoriale
 - Accesarea atributelor
 - Parcurgerea entităților selectate
 - Parcurgerea unui subset de entități
- Modificarea straturilor vectoriale
 - Adăugarea entităților
 - Ștergerea entităților
 - Modificarea entităților
 - Adăugarea și eliminarea câmpurilor
- Modificarea straturi vectoriale prin editarea unui tampon de memorie
- Crearea unui index spațial
- Scrierea straturilor vectoriale
- Furnizorul de memorie
- Aspectul (simbologia) straturilor vectoriale
 - Render cu Simbol Unic
 - Render cu Simboluri Categorisite
 - Render cu Simboluri Graduale
 - Lucrul cu Simboluri
 - * Lucrul cu Straturile Simbolului
 - * Crearea unor Tipuri Personalizate de Straturi pentru Simboluri
 - Crearea renderelor Personalizate
- Lecturi suplimentare

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

5.1 Obținerea informațiilor despre atribute

Puteți obține informațiile despre câmpurile asociate cu un strat vectorial, prin apelarea `pendingFields()` pe o instanță `QgsVectorLayer`:

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

Note: De asemenea, începând de la QGIS 2.12 există și o funcție `fields()` în `QgsVectorLayer`, care este un alias pentru `pendingFields()`.

5.2 Selectarea entităților

În QGIS, entitățile pot fi selectate în diverse moduri, utilizatorul putând efectua clic pe o entitate, pentru a trasa un dreptunghi pe canvașul hărții sau pentru a folosi o expresie de filtrare. Entitățile selectate sunt în mod normal evidențiate printr-o culoare diferită (galben, în mod implicit), pentru a atrage atenția utilizatorului asupra selecției. Uneori poate fi util să selectați programatic entitățile sau să schimbați culoarea implicită.

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

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

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

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

Pentru a anula selecția, transmiteți doar o listă vidă:

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

5.3 Iterații în straturile vectoriale

Parcursirea elementelor dintr-un strat vectorial este una dintre cele mai obișnuite activități. Mai jos este prezentat un exemplu de cod de bază, simplu, pentru a efectua această sarcină și care arată unele informații despre fiecare entitate spațială. Variabila `layer` se consideră a conține un obiect `QgsVectorLayer`

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

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

    # fetch attributes
    attrs = feature.attributes()

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


5.3.1 Accesarea atributelor

Atributele pot fi apelate după numele lor.

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

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

```
print feature[0]
```

5.3.2 Parcurgerea entităților selectate

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

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

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

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

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

5.3.3 Parcurgerea unui subset de 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 feature in layer.getFeatures(request):
    # do whatever you need with the feature
```

Dacă aveți nevoie de un filtru pe bază de atribut în locul unuia spațial (sau în plus față de acesta), așa cum se vede în exemplul de mai sus, puteți construi un obiect `QgsExpression` și să-i transmiteți constructorul `QgsFeatureRequest`. Iată un exemplu

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

Vedeți *Expresii, filtrarea și calculul valorilor* pentru detalii despre sintaxa acceptată de `QgsExpression`.

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

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

Tip: Dacă aveți nevoie doar de un subset de atribute sau dacă nu aveți nevoie de informațiile geometrice, puteți crește în mod semnificativ **viteza** cererii entităților, prin utilizarea fanionului `QgsFeatureRequest.NoGeometry`, sau specificând un subset de atribute (eventual vid), așa cum s-a arătat în exemplul de mai sus.

5.4 Modificarea straturilor vectoriale

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

```
caps = layer.dataProvider().capabilities()
# Check if a particular capability is supported:
caps & QgsVectorDataProvider.DeleteFeatures
# Print 2 if DeleteFeatures is supported
```

Pentru o listă a tuturor capacităților disponibile, vă rugăm să consultați [Documentația API pentru QgsVectorDataProvider](#)

Pentru a genera, într-o listă separată prin virgule, descrierea capacităților stratului, puteți folosi `capabilitiesString()`, ca în exemplul următor:

```
caps_string = layer.dataProvider().capabilitiesString()
# Print:
# u'Add Features, Delete Features, Change Attribute Values,
# Add Attributes, Delete Attributes, Create Spatial Index,
# Fast Access to Features at ID, Change Geometries,
# Simplify Geometries with topological validation'
```

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 efectuarea *modifications with editing buffer*.

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

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

5.4.1 Adăugarea entităților

Creați câteva instanțe ale clasei `QgsFeature` și transmiteți o listă a acestora furnizorului `addFeatures()`. Acesta va returna două valori: rezultatul (`true/false`) și lista entităților adăugate (ID-urile lor fiind stabilite de către depozitul de date).

Pentru a configura atributele, puteți fie să inițializați entitatea, transmitând o instanță a clasei `QgsFields`, fie să apelați `initAttributes()` cu numărul de câmpuri pe care doriți să le adăugați.

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.pendingFields())
    feat.setAttributes([0, 'hello'])
    # Or set a single attribute by key or by index:
    feat.setAttribute('name', 'hello')
```

```
feat.setAttribute(0, 'hello')
feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
(res, outFeats) = layer.dataProvider().addFeatures([feat])
```

5.4.2 Ștergerea entităților

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

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

5.4.3 Modificarea entităților

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

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

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

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

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

5.4.4 Adăugarea și eliminarea câmpurilor

Pentru a adăuga câmpuri (atribute), trebuie să specificați o listă de definiții pentru acestea. Pentru ștergerea de câmpuri e suficientă furnizarea unei liste de indecși pentru câmpuri.

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

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

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

```
layer.updateFields()
```

5.5 Modificarea straturi vectoriale prin editarea unui tampon de memorie

Când editați vectori în aplicația QGIS, în primul rând, trebuie să comutați în modul de editare pentru stratul în care lucrați, apoi să efectuați modificări pe care, în cele din urmă, să le salvați (sau să le anulați). 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. Utilizați această opțiune atunci

când furnizați unele instrumente GUI pentru editarea straturilor vectoriale, permițând utilizatorului să decidă dacă să salveze/anuleze, și punându-i la dispoziție facilitățile de undo/redo. Atunci când salvați modificările, acestea vor fi transferate din memoria tampon de editare în furnizorul de date.

Pentru a afla dacă un strat se află în modul de editare, utilizați `isEditable()` — funcțiile de editare funcționând numai atunci când modul de editare este activat. Utilizarea funcțiilor de editare

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

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

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

Pentru ca undo/redo să funcționeze în mod corespunzător, apelurile de mai sus trebuie să fie înglobate în comenzi undo. (Dacă nu vă pasă de undo/redo și doriți să stocați imediat modificările, atunci veți avea o sarcină mai ușoară prin *editing with data provider*.) Cum să utilizați funcționalitatea undo

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

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

if problem_occurred:
    layer.destroyEditCommand()
    return

# ... more editing ...

layer.endEditCommand()
```

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

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

De asemenea, puteți utiliza expresia `with edit(layer)` - pentru a încorpora într-un bloc de cod semantic, pentru `commit` și `rollback`, așa cum se arată în exemplul de mai jos:

```
with edit(layer):
    f = layer.getFeatures().next()
    f[0] = 5
    layer.updateFeature(f)
```

La final, se va apela în mod automat funcția `commitChanges()`. În cazul în care se produce o excepție, toate modificările vor fi anulate `rollback()`. În cazul unei probleme în cadrul `commitChanges()` (atunci când metoda returnează valoarea `False`) va apărea o excepție `QgsEditError`.

5.6 Crearea unui index spațial

Indecșii spațiali pot îmbunătăți dramatic performanța codului dvs, în cazul în care este nevoie să interogați frecvent un strat vectorial. Imaginați-vă, de exemplu, că scrieți un algoritm de interpolare, și că, pentru o anumită locație, trebuie să aflați cele mai apropiate 10 puncte dintr-un strat, în scopul utilizării acelor puncte în calculul valorii interpolate. Fără un index spațial, singura modalitate pentru QGIS de a găsi cele 10 puncte, este de a calcula distanța tuturor punctelor față de locația specificată și apoi de a compara aceste distanțe. Această sarcină poate fi mare consumatoare de timp, mai ales în cazul în care trebuie să fie repetată pentru mai multe locații. Dacă pentru stratul respectiv există un index spațial, operațiunea va fi mult mai eficientă.

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

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

- crearea index spațial — următorul cod creează un index vid

```
index = QgsSpatialIndex()
```

- să adăugați entitățile la index — Indexul preia obiectul `QgsFeature`, iar apoi la structura internă de date. Puteți crea obiectul manual sau să folosiți unul dintre apelurile anterioare către funcția `nextFeature()` a furnizorului

```
index.insertFeature(feats)
```

- o dată ce ați introdus valori în indexul spațial, puteți efectua unele interogări

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

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

5.7 Scrierea straturilor vectoriale

Puteți scrie în fișierele conținând straturi vectoriale folosind clasa `QgsVectorFileWriter`. Aceasta acceptă orice alt tip de fișier vector care suportă OGR (fișiere shape, GeoJSON, KML și altele).

Există două posibilități de a exporta un strat vectorial:

- dintr-o instanță a `QgsVectorLayer`

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

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

error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSON"
if error == QgsVectorFileWriter.NoError:
    print "success again!"
```

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

For valid driver names please consult the 'supported formats by OGR' --- you should pass the value in the "Code" column as the driver name. Optionally

you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes --- look into the documentation for full syntax.

- direct din entități

```
# define fields for feature attributes. A QgsFields object is needed
fields = QgsFields()
fields.append(QgsField("first", QVariant.Int))
fields.append(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 WKBTYPe enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Shapefile")

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

# 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
del writer
```

5.8 Furnizorul de memorie

Furnizorul de memorie este destinat, în principal, dezvoltatorilor de plugin-uri sau de aplicații terț. El nu stochează date pe disc, permițând 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, veți fi capabili de a parcurge mai rapid entitățile, în interiorul unor regiuni mai mici (din moment ce nu este necesar să traversați toate entitățile, ci doar pe cele din dreptunghiul specificat).

Un furnizor de memorie este creat prin transmiterea "memoriei" ca șir 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 Specificați sistemul de referință de coordonate, unde definiția poate fi oricare din formele acceptate de: `QgsCoordinateReferenceSystem.createFromString()`

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

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

Următorul exemplu de URI încorporează toate aceste opțiuni

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

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

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

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

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

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

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

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

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

5.9 Aspectul (simbologia) straturilor vectoriale

Când un strat vector este randat, aspectul datelor este dat de **render** și 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()
```

Și cu acea referință, să explorăm un pic

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

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

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

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

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

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

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

5.9.1 Render cu Simbol Unic

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

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

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

De exemplu, puteți schimba simbolul folosit de un strat particular de tip **punct**, prin apelarea `setSymbol()`, transmițându-i o instanță `QgsMarkerSymbolV2`, ca în următorul exemplu de cod:

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

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

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

- vârful_de_săgeată_plin
- x

Pentru a obține lista completă de proprietăți, pentru primul strat simbol al unei instanțe, puteți urmări exemplul de cod:

```
print layer.rendererV2().symbol().symbolLayers()[0].properties()
# Prints
{u'angle': u'0',
u'color': u'0,128,0,255',
u'horizontal_anchor_point': u'1',
u'name': u'circle',
u'offset': u'0,0',
u'offset_map_unit_scale': u'0,0',
u'offset_unit': u'MM',
u'outline_color': u'0,0,0,255',
u'outline_style': u'solid',
u'outline_width': u'0',
u'outline_width_map_unit_scale': u'0,0',
u'outline_width_unit': u'MM',
u'scale_method': u'area',
u'size': u'2',
u'size_map_unit_scale': u'0,0',
u'size_unit': u'MM',
u'vertical_anchor_point': u'1'}
```

Acest lucru poate fi util dacă doriți să modificați unele proprietăți:

```
# You can alter a single property...
layer.rendererV2().symbol().symbolLayer(0).setName('square')
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.rendererV2().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.rendererV2().setSymbol(QgsMarkerSymbolV2.createSimple(props))
```

5.9.2 Render cu Simboluri Categorisite

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

Pentru a obține o listă de categorii

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

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

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

5.9.3 Render cu Simboluri Graduale

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

Pentru a afla mai multe despre gamele utilizate în render

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

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.

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

```
from qgis.core import *

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

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

5.9.4 Lucrul cu Simboluri

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

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

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

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

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

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

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

Lucrul cu Straturile Simbolului

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

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

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

Rezultat

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

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

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

Crearea unor Tipuri Personalizate de Straturi pentru Simboluri

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

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

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

    def layerType(self):
```

```
    return "FooMarker"

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

def startRender(self, context):
    pass

def stopRender(self, context):
    pass

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

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

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

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

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

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

        self.layer = None

        # setup a simple UI
        self.label = QLabel("Radius:")
        self.spinRadius = QDoubleSpinBox()
        self.hbox = QHBoxLayout()
        self.hbox.addWidget(self.label)
        self.hbox.addWidget(self.spinRadius)
        self.setLayout(self.hbox)
        self.connect(self.spinRadius, SIGNAL("valueChanged(double)"), \
            self.radiusChanged)

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

    def symbolLayer(self):
        return self.layer

    def radiusChanged(self, value):
        self.layer.radius = value
```

```
self.emit(SIGNAL("changed()"))
```

Acest widget poate fi integrat în fereastra de proprietăți a simbolului. În cazul în care tipul de strat simbol este selectat în fereastra de proprietăți a simbolului, se creează o instanță a stratului simbol și o instanță a widget-ului stratului simbol. Apoi, se apelează metoda `setSymbolLayer()` pentru a 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-și 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 fișiere de proiect cu straturi simbol personalizate sau incapacitatea de a edita atributele stratului în GUI.

Va trebui să creăm metadate pentru stratul simbolului

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

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

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

    def createSymbolLayerWidget(self):
        return FooSymbolLayerWidget()
```

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

Ar trebui să 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 — și am încheiat.

5.9.5 Crearea renderelor Personalizate

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

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

```
import random

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

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

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

```
s.startRender(context)

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

def usedAttributes(self):
    return []

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

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

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

```
class RandomRendererWidget(QgsRendererV2Widget):
    def __init__(self, layer, style, renderer):
        QgsRendererV2Widget.__init__(self, layer, style)
        if renderer is None or renderer.type() != "RandomRenderer":
            self.r = RandomRenderer()
        else:
            self.r = renderer
        # setup UI
        self.btn1 = QPushButtonV2()
        self.btn1.setColor(self.r.syms[0].color())
        self.vbox = QVBoxLayout()
        self.vbox.addWidget(self.btn1)
        self.setLayout(self.vbox)
        self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

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

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

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

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

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

    def createRenderer(self, element):
        return RandomRenderer()

    def createRendererWidget(self, layer, style, renderer):
```

```
return RandomRendererWidget(layer, style, renderer)
```

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

În mod similar cu straturile simbol, constructorul de 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.

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

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

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

5.10 Lecturi suplimentare

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

Manipularea geometriei

- Construirea geometriei
- Accesarea geometriei
- Predicate și operațiuni geometrice

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-părți). 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 țară formată din mai multe insule poate fi reprezentată ca un multi-poligon.

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

6.1 Construirea geometriei

Există mai multe opțiuni pentru a crea o geometrie:

- din coordonate

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

Coordonatele sunt obținute folosind clasa `QgsPoint`.

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

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

- din well-known text (WKT)

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

- din well-known binary (WKB)

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

6.2 Accesarea geometriei

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

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

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

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

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

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

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

6.3 Predicate și operațiuni geometrice

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

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

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

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

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

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

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

Informații suplimentare pot fi găsite în sursele următoare:

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

Proiecții suportate

- Sisteme de coordonate de referință
- Proiecții

7.1 Sisteme de coordonate de referință

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

- specifică CRS-ul după ID-ul său

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

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

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

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

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

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

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

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

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

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

Accesarea informațiilor sistemului de referință spațial

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

7.2 Proiecții

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

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

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

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

Folosirea suportului de hartă

- Încapsularea suportului de hartă
- Folosirea instrumentelor în suportul de hartă
- Benzile elastice și marcajele nodurilor
- Dezvoltarea instrumentelor personalizate pentru suportul de hartă
- Dezvoltarea elementelor personalizate pentru suportul de hartă

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

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

Ori de câte ori harta a fost deplasată, mărită/micșorată (sau alte acțiuni care declanșează o recitare), harta este randată iarăși în interiorul granițelor 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 afișarea hărții este `QgsMapCanvasMap`. Această clasă controlează, de asemenea, recitirea hărții randate. În afară de acest element, care acționează ca fundal, pot exista mai multe **elemente ale suportului hărții**. Elementele tipice suportului de hartă sunt benzile elastice (utilizate pentru măsurare, editare vectorială etc) sau marcajele 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 corespunzător creează o bandă elastică de forma actuală a poligonului. Toate elementele suportului de hartă reprezintă subclase ale `QgsMapCanvasItem` care adaugă mai multe funcționalități obiectelor de bază `QGraphicsItem`.

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

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

8.1 Încapsularea suportului de hartă

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

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

Acest cod va produce o fereastră de sine stătătoare cu suport pentru hartă. Ea poate fi, de asemenea, încorporată într-un widget sau într-o fereastră deja existente. Atunci când se utilizează fișiere .ui și 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 fișier 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 creați o așezare în pagină.

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

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

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

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

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

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

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

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

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

8.2 Folosirea instrumentelor în suportul de hartă

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

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

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

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

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

        self.setCentralWidget(self.canvas)
```



```

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

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

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

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

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

self.pan()

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

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

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

```

Puteți pune codul de mai sus într-un fișier, de exemplu, `mywnd.py` și să-l încercați apoi în consola Python din QGIS. Acest cod va pune stratul curent selectat în noul canvas creat

```

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

```

Doar 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/ma')` — altfel declarația de import nu va reuși, negăsind modulul.

8.3 Benzile elastice și marcajele nodurilor

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

Pentru a afișa o polilinie

```

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

```

Pentru a afișa un poligon

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

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 granița exterioară, în plus (opțional) inelele corespund găurilor din poligon.

Benzile elastice acceptă unele personalizări, și anume schimbarea culorii și a lățimii liniei

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

Elementele suportului sunt legate de suportul hărții. Pentru a le ascunde temporar (și a le arăta din nou, folosiți combinația `hide()` și `show()`). Pentru a elimina complet elementul, trebuie să-l eliminăm de pe scena canevaului

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

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

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

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

În acest mod se va desena o cruciuliță roșie pe poziția [0,0]. Este posibilă personalizarea tipului pictogramei, dimensiunea, culoarea și lățimea instrumentului de desenare

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

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

8.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă

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

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.

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

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

```

self.rubberBand.setWidth(1)
self.reset()

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

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

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

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

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

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

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

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

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

    return QgsRectangle(self.startPoint, self.endPoint)

def deactivate(self):
    super(RectangleMapTool, self).deactivate()
    self.emit(SIGNAL("deactivated()"))

```

8.5 Dezvoltarea elementelor personalizate pentru suportul de hartă

DE EFECTUAT: how to create a map canvas item

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

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

def show_canvas(app):
    canvas = QgsMapCanvas()
    canvas.show()
    app.exec_()
app = init()
show_canvas(app)
```

Randarea hărților și imprimarea

- Randarea simplă
- Randarea straturilor cu diferite CRS-uri
- Generarea folosind Compozitorul de hărți
 - Ieșire ca imagine raster
 - Ieșire în format PDF

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 ieșiri mai rafinate, prin compunerea hărții cu ajutorul clasei `QgsComposition`.

9.1 Randarea simplă

Randați mai multe straturi, folosind `QgsMapRenderer` — creați destinația dispozitivului de colorare (`QImage`, `QPainter` etc), setați stratul, limitele sale, dimensiunea de ieșire și efectuați randarea

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

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

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

render = QgsMapRenderer()

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

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

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

# do the rendering
render.render(p)
```

```
p.end()

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

9.2 Randarea straturilor cu diferite CRS-uri

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

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

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

9.3 Generarea folosind Compozitorul de hărți

Compozitorul de hărți reprezintă un instrument foarte util în cazul în care doriți să elaborați ceva mai sofisticat decât simpla randare de mai sus. Utilizând Constructorului este posibilă crearea unor machete complexe de hărți, 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.

Compozitorul constă într-o serie de clase. Toate acestea fac parte din biblioteca de bază. Aplicația QGIS are un GUI convenabil pentru plasarea elementelor, deși nu face parte din biblioteca GUI. Dacă nu sunteți familiarizați cu [Cadru de lucru Qt Graphics View](#), atunci vă încurajăm să verificați documentația acum, deoarece compozitorul este bazat pe el. De asemenea, verificați [documentația Python de implementare a QGraphicsView](#).

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

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

Rețineți: 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 utilizați compozitorul într-o aplicație independentă, vă puteți crea propria dvs. instanță de render de hărți, în același mod cum s-a arătat în secțiunea de mai sus, și să-l transmiteți compoziției.

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

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

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

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

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

- legenda

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

- scara grafică

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

- săgeată
- imagine
- formă
- tabelă

În mod implicit, elementele compozitorului nou creat au poziția zero (colțul din stânga sus a paginii) și dimensiunea zero. Poziția și dimensiunea sunt măsurate întotdeauna în milimetri

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

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

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

Pe lângă crearea manuală a elementele compozitorului, QGIS are suport pentru șabloane, 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.

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

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

9.3.1 Ieșire ca imagine raster

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

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

# create output image and initialize it
image = QImage(QSize(width, height), QImage.Format_ARGB32)
```

```
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

# render the composition
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()

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

9.3.2 Ieșire în format PDF

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

```
printer = QPainter()
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

- Parsarea expresiilor
- Evaluarea expresiilor
 - Expresii de bază
 - Expresii cu entități
 - Tratarea erorilor
- Exemple

QGIS are un oarecare suport pentru analiza expresiilor, cum ar fi SQL. Doar un mic subset al sintaxei SQL este acceptat. Expresiile pot fi evaluate fie ca predicate booleene (returnând True sau False), fie ca funcții (care întorc o valoare scalară). Parcurgeți *vector_expressions* din Manualul Utilizatorului, pentru o listă completă a funcțiilor disponibile.

Trei tipuri de bază sunt acceptate:

- — număr atât numere întregi cât și numere zecimale, de exemplu, 123, 3.14
- șir — 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 forțarea priorității operatorului: (1 + 1) * 3
- plus și minus unari: -12, +5
- funcții matematice: sqrt, sin, cos, tan, asin, acos, atan
- funcții de conversie: to_int, to_real, to_string, to_date
- funcții geometrice: \$area, \$length
- funcții de manipulare a geometriei: \$x, \$y, \$geometry, num_geometries, centroid

Și următoarele predicate sunt suportate:

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

Exemple de predicate:

- 1 + 2 = 3
- sin(angle) > 0

- 'Hello' LIKE 'He%'
- (x > 10 AND y > 10) OR z = 0

Exemple de expresii scalare:

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

10.1 Parsarea expresiilor

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

10.2 Evaluarea expresiilor

10.2.1 Expresii de bază

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

10.2.2 Expresii cu entități

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

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

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

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

10.2.3 Tratarea erorilor

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

value = exp.evaluate()
```

```
if exp.hasEvalError():
    raise ValueError(exp.evalErrorString())

print value
```

10.3 Exemple

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

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

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

Citirea și 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 și QGIS API. Pentru fiecare variabilă ar trebui să alegeți o cheie care va fi folosită pentru a accesa variabila — pentru culoarea preferată a utilizatorului ați putea folosi o cheie de genul “culoare_favorită” sau orice alt șir semnificativ. Este recomandabil să folosiți o oarecare logică în denumirea cheilor.

Putem face diferența între mai multe tipuri de setări:

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

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

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

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

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

```
proj = QgsProject.instance()

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

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

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

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

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

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

Comunicarea cu utilizatorul

- Afișarea mesajelor. Clasa `QgsMessageBar`
- Afișarea progresului
- Jurnalizare

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

12.1 Afișarea mesajelor. Clasa `QgsMessageBar`

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

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

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

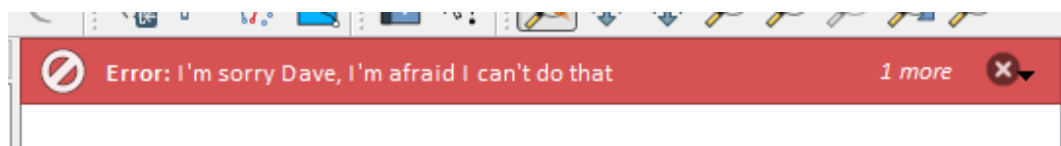


Figure 12.1: Bara de mesaje a QGIS

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

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

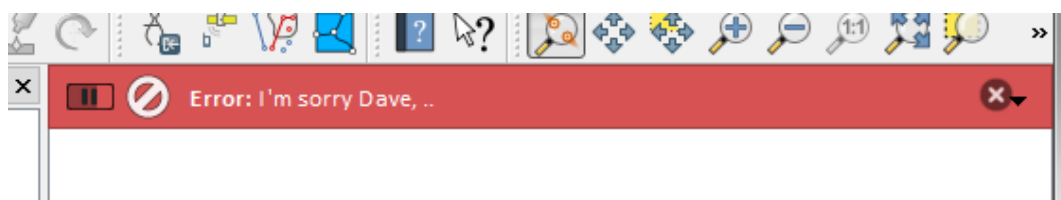


Figure 12.2: Bara de mesaje a QGIS, cu cronometru

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

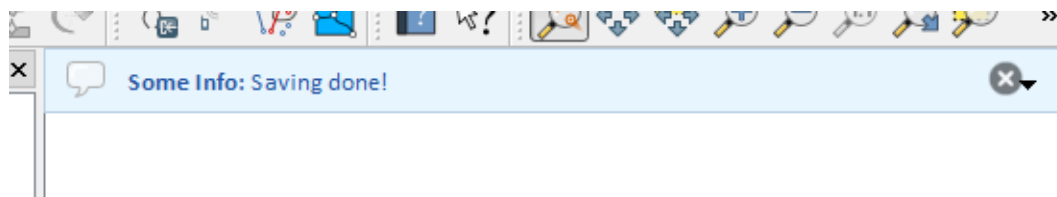


Figure 12.3: Bara de mesaje a QGIS (info)

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

```
def showError():
    pass

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

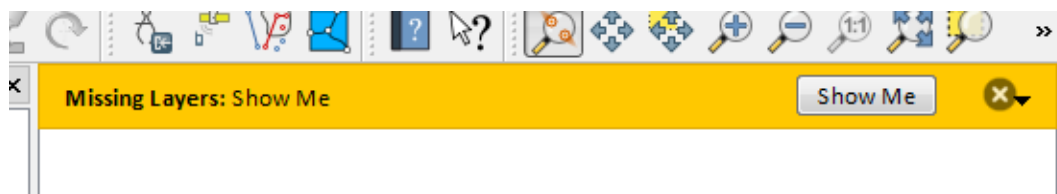


Figure 12.4: Bara de mesaje a QGIS, cu un buton

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

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

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

12.2 Afișarea progresului

Barele de progres pot fi, de asemenea, incluse în bara de mesaje QGIS, din moment ce, așa 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()
```

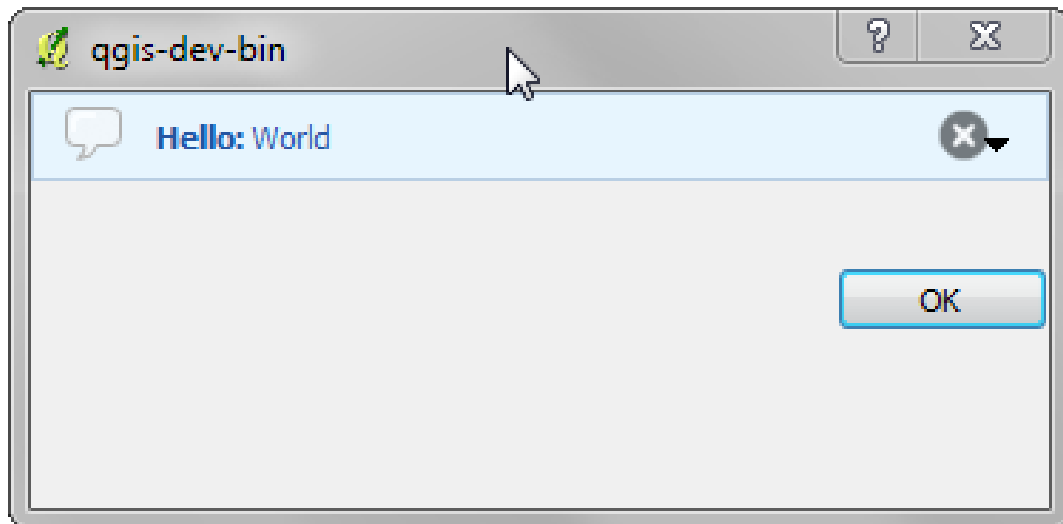



Figure 12.5: Bara de mesaje a QGIS, într-o fereastră de dialog

```

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()

```

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

```

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

```

12.3 Jurnalizare

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

```

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

```

Dezvoltarea plugin-urilor Python

- Scrierea unui plugin
 - Fișierele Plugin-ului
- Conținutul Plugin-ului
 - Metadatele plugin-ului
 - `__init__.py`
 - `mainPlugin.py`
 - Fișier de resurse
- Documentație
- Traducerea
 - Cerințe software
 - Fișierele și directorul
 - * fișier `.pro`
 - * fișier `.ts`
 - * fișier `.qm`
 - Încărcarea plugin-ului

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

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

- UNIX/Mac: `~/ .qgis2/python/plugins` și `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/ .qgis2/python/plugins` și `(qgis_prefix)/python/plugins`

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

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

Pași:

1. *Ideea:* Conturați-vă o idee despre ceea ce vreți să faceți cu noul plugin QGIS. De ce-l faceți? Ce problemă doriți să rezolve? Există deja un alt plugin pentru această problemă?
2. *Creare fișiere:* Se creează fișierele descrise în continuare. Un punct de plecare (`__init__.py`). Completați *Metadatele plugin-ului* (`metadata.txt`). Un corp python principal al plugin-ului (`mainplugin.py`). Un formular în QT-Designer (`form.ui`), cu al său `resources.qrc`.
3. *Scrierea codului:* Scrieți codul în interiorul `mainplugin.py`

4. *Testul*: Închideți și re-deschideți QGIS, apoi importați-l din nou. Verificați dacă totul este în regulă.
5. *Publicare*: Se publică plugin-ul în depozitul QGIS sau vă faceți propriul depozit ca un “arsenal” de “arme GIS” personale

13.1 Scrierea unui plugin

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

13.1.1 Fișierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin

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

Care este semnificația fișierelor:

- `__init__.py` = Punctul de plecare al plugin-ului. Acesta trebuie să aibă metoda `classFactory()` și poate avea orice alt cod de inițializare.
- `mainPlugin.py` = Principalul cod lucrativ al plugin-ului. Conține toate informațiile cu privire la acțiunile plugin-ului și ale codului principal.
- `resources.qrc` = Documentul .xml creat de Qt Designer. Conține căi relative la resursele formelor.
- `resources.py` = Traducerea în Python a fișierului .qrc, descris mai sus.
- `form.ui` = GUI-ul creat de Qt Designer.
- `form.py` = Traducerea în Python a fișierului form.ui, descris mai sus.
- `metadata.txt` = Necesară pentru QGIS >= 1.8.0. Conține informații generale, versiunea, numele și alte metadate utilizate de către site-ul de plugin-uri și de către infrastructura plugin-ului. Începând cu QGIS 2.0 metadatele din `__init__.py` nu mai sunt acceptate, iar `metadata.txt` este necesar.

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

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

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

13.2 Conținutul Plugin-ului

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

13.2.1 | Metadatele plugin-ului

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

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

Numele metadatei	Obligato-riu	Note
<code>nume</code>	True	un șir scurt conținând numele pluginului
<code>qgisMinimumVersion</code>	True	notație cu punct a versiunii minime QGIS
<code>qgisMaximumVersion</code>	False	notație cu punct a versiunii maxime QGIS
<code>descriere</code>	True	scurt text care descrie plugin-ul, HTML nefiind permis
<code>despre</code>	True	text mai lung care descrie plugin-ul în detalii, HTML nefiind permis
<code>versiune</code>	True	scurt șir cu versiunea notată cu punct
<code>autor</code>	True	nume autor
<code>email</code>	True	e-mail-ul autorului, care nu este afișat în managerul de plugin-uri QGIS sau pe site-ul web, fiind vizibile doar pentru utilizatorii autentificați, cum ar fi alți autori de plugin-uri și administratorii site-ului
<code>jurnalul schimbărilor</code>	False	șir, poate fi pe mai multe linii, HTML nefiind permis
<code>experimental</code>	False	semnalizator boolean, <i>True</i> sau <i>False</i>
<code>învechit</code>	False	semnalizator boolean, <i>True</i> sau <i>False</i> , se aplică întregului plugin și nu doar la versiunea încărcată
<code>etichete</code>	False	o listă de valori separate prin virgulă, spațiile fiind permise în interiorul etichetelor individuale
<code>pagina de casă</code>	False	o adresă URL validă indicând spre pagina plugin-ului dvs.
<code>depozit</code>	True	o adresă URL validă pentru depozitul de cod sursă
<code>tracker</code>	False	o adresă validă pentru bilete și rapoartare de erori
<code>pictogramă</code>	False	un nume de fișier sau o cale relativă (relativă la directorul de bază al pachetului comprimat al plugin-ului) pentru o imagine adecvată pentru web (PNG, JPEG)
<code>categorii</code>	False	una din valorile <i>Raster</i> , <i>Vector</i> , <i>Bază de date</i> și <i>Web</i>

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

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

```
category=Raster
```

Note: În cazul în care valoarea `qgisMaximumVersion` este vidă, ea va fi setată automat la versiunea majoră plus `.99` încărcată în depozitul *Depozitul oficial al plugin-urilor python*.

Un exemplu pentru acest `metadata.txt`

```
; the next section is mandatory
```

```
[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
        lines starting with spaces belong to the same
        field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
; 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
icon=icon.png

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

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

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

13.2.2 `__init__.py`

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

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

## any other initialisation needed
```

13.2.3 mainPlugin.py

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

```

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

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

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

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

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

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

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

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

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

```

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

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

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

- `addPluginToRasterMenu()`
- `addPluginToVectorMenu()`

- `addPluginToDatabaseMenu()`
- `addPluginToWebMenu()`

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

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

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

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

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

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

Nu uitați să stabiliți pentru `QAction` și `QMenu` `objectName` un nume specific plugin-ului dvs, astfel încât acesta să poată fi personalizat.

13.2.4 Fișier de resurse

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

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

Este bine să folosiți un prefix pentru a evita coliziunile cu alte plugin-uri sau cu oricare alte părți ale QGIS, în caz contrar, ați putea obține rezultate nedorite. Trebuie doar să generați un fișier Python care va conține resursele. Acest lucru se face cu comanda: **:comanda:‘pyrcc4’**

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

Note: În mediile Windows, încercând să rulați `pyrcc4` din Linia de Comandă sau din Powershell va rezulta, probabil, eroarea “Windows cannot access the specified device, path, or file [...]”. Cea mai simplă soluție constă, probabil, în utilizarea aplicației OSGeo4W, dar dacă puteți modifica variabila de mediu `PATH`, sau să specificați calea către executabilul explicit, ar trebui să-l găsiți în `<Your QGIS Install Directory>\bin\pyrcc4.exe`.

Și asta e tot ... nimic complicat :)

Dacă ați făcut totul corect, ar trebui să găsiți și să încărcați plugin-ul în managerul de plugin-uri și să vedeți un mesaj în consolă, atunci când este selectat meniul adecvat sau pictograma din bara de instrumente.

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.

13.3 Documentație

Documentația pentru plugin poate fi scrisă ca fișiere 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`, afișând ceea ce găsește mai întâi. Aici, `ll_cc` reprezintă limba în care se afișează 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 afișat ajutorul, numele de fișier, care poate înlocui 'index' în numele fișierelor în care se caută, și secțiunea, care este numele unei ancore HTML în documentul în care se va poziționa browser-ul.

13.4 Traducerea

Cu câțiva pași puteți configura mediul pentru localizarea plugin-ului, astfel încât, în funcție de setările regionale ale computerului, plugin-ul va fi încărcat în diferite limbi.

13.4.1 Cerințe software

Cel mai simplu mod de a crea și de a gestiona toate fișierele de traducere este de a instala [Qt Linguist](#). Într-un mediu precum Linux îl puteți instala tastând:

```
sudo apt-get install qt4-dev-tools
```

13.4.2 Fișierele și directorul

Atunci când creați plugin-ul, veți găsi folderul `i18n` în directorul principal al plugin-urilor.

Toate fișierele de traducere trebuie să se afle în acest director.

fișier .pro

Mai întâi, ar trebui să creați un fișier `.pro`, care este un fișier *proiect*, gestionabil de către Qt Linguist.

În acest fișier `.pro` trebuie să specificați toate fișierele și formularele pe care doriți să le traduceți. Acest fișier este folosit pentru a configura fișierele de localizare și variabilele. Un exemplu de fișier `pro`:

```
FORMS = ../ui/*

SOURCES = ../your_plugin.py

TRANSLATIONS = your_plugin_it.ts
```

În acest caz particular, toate interfețele cu utilizatorul sunt plasate în folderul `../ui`, iar dvs. doriți să le traduceți pe toate.

Mai mult, fișierul `dvs.your_plugin.py` este cel care *apelează* toate meniurile și sub-meniurile plugin-ului dvs. din bara de instrumente QGIS, iar dumneavoastră doriți să le traduceți pe toate.

În cele din urmă, cu ajutorul variabilei `TRANSLATIONS` puteți specifica limbile de traducere pe care le doriți.

Warning: Asigurați-vă că denumirea fișierului `ts` este ceva de genul `pluginul_dvs + limba + .ts`, în caz contrar încărcarea limbii va eșua! Utilizați o prescurtare pe 2 litere a limbii (**it** pentru italiană, **de** pentru germană etc...)

fișier .ts

O dată ce ați creat un `.pro` sunteți gata să generați fișier(e) `.ts`, ale limb(ilor) plugin-ului dvs.

Deschideți o fereastră terminal, mergeți în directorul `your_plugin/i18n` și introduceți:

```
lupdate your_plugin.pro
```

ar trebui să vedeți fișier(e) `your_plugin_language.ts`.

Deschideți cu **Qt Linguist** fișierul `.ts`, apoi începeți să-l traduceți,

fișier .qm

Când ați terminat de tradus plugin-ul (în cazul în care unele șiruri de caractere nu sunt completate, pentru acestea va fi utilizată limba sursă) trebuie să creați un fișier `.qm` (fișierul `.ts`, compilat, care va fi utilizat de către QGIS).

Este suficient să deschideți o fereastră terminal în directorul `your_plugin/i18n`, apoi tastați:

```
lrelease your_plugin.ts
```

acum, în directorul `i18n` veți vedea fișier(e) `pluginului_dvs.qm`.

13.4.3 Încărcarea plugin-ului

Pentru a putea vedea traducerea plugin-ului dvs., este suficient să deschideți QGIS, schimbați limba (*Setări* → *Opțiuni* → *Limba*) și restartați QGIS.

Ar trebui să vedeți plugin-ul în limba corectă.

Warning: Dacă schimbați ceva în plugin (noi interfețe, un meniu nou etc.), va trebui să **generați din nou** versiunea de actualizare a fișierelor `.ts` și `.qm`, de aceea, executați iarăși comanda de mai sus.

Setările IDE pentru scrierea și depanarea de plugin-uri

- O notă privind configurarea IDE-ului în Windows
- Depanare cu ajutorul Eclipse și PyDev
 - Instalare
 - Pregătirea QGIS
 - Configurarea Eclipse
 - Configurarea depanatorului
 - Configurați Eclipse pentru a înțelege API-ul
- Depanarea cu ajutorul PDB

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

14.1 O notă privind configurarea IDE-ului în Windows

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

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

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

- Faceți o copie a fișierului `qgis-unstable.bat` și redenumiți-o `pyscripter.bat`.
- Deschideți-o într-un editor. Apoi eliminați ultima linie, cea care startează QGIS.
- Adăugați o linie care să indice calea către executabilul Pyscripter, apoi adăugați argumentul care stabilește versiunea de Python ce urmează a fi utilizată (2.7 în cazul QGIS >= 2.0)
- De asemenea, adăugați și un argument care să indice folderul unde poate găsi Pyscripter dll-ul Python folosit de către QGIS, acesta aflându-se în folderul `bin` al instalării OSGeoW

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

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

Mult mai popular decât Pyscripter, Eclipse este o alegere comună în rândul dezvoltatorilor. În următoarele secțiuni, vă vom explica cum să-l configurați pentru dezvoltarea și testarea plugin-urilor. Pentru utilizarea în Windows, ar trebui să creați, de asemenea, un fișier batch pe care să-l utilizați la pornirea Eclipse.

Pentru a crea fișierul batch, urmați acești pași:

- Localizați folderul în care rezidă `qgis_core.dll`. În mod normal, el se găsește în `C:\OSGeo4W\apps\qgis\bin`, dar dacă ați compilat propria aplicație QGIS, atunci el va fi în folderul `output/bin/RelWithDebInfo`
- Localizați executabilul `eclipse.exe`.
- Creați următorul script și folosiți-l pentru a starta Eclipse, atunci când dezvoltați plugin-uri QGIS.

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

14.2 Depanare cu ajutorul Eclipse și PyDev

14.2.1 Instalare

Pentru a utiliza Eclipse, asigurați-vă că ați instalat următoarele

- Eclipse
- Plugin-ul Eclipse Aptana sau PyDev
- QGIS 2.x

14.2.2 Pregătirea QGIS

Chiar și în QGIS, este necesară efectuarea anumitor acțiuni pregătitoare. Două plugin-uri sunt de interes: *Remote Debug* și *Plugin Reloader*.

- Mergeți la *Plugin-uri* → *Gestiune și Instalare plugin-uri...*
- Căutați *Remote Debug* (la această dată este încă experimental, deci, în cazul în care nu-l observați, va trebui să activați plugin-urile experimentale, din fila *Opțiunilor*). Instalați-l.
- De asemenea, căutați *Plugin Reloader* și instalați-l. Acest lucru vă va permite să reîncărcați un plug-in, fără a fi necesare închiderea și repornirea QGIS.

14.2.3 Configurarea Eclipse

În Eclipse, creați un nou proiect. Puteți să selectați *General Project* și să legați ulterior sursele dvs. reale, așa că nu prea contează unde plasați acest proiect.

Acum, faceți clic dreapta pe noul proiect și alegeți *New* → *Folder*.

Faceți clic pe [**Avansat**] și alegeți *Legătură către o locație alternativă (Folder Legat)*. În cazul în care deja aveți sursele pe care doriți să le depanați, le puteți alege pe acestea. În caz contrar, creați un folder, așa cum s-a explicat anterior.

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

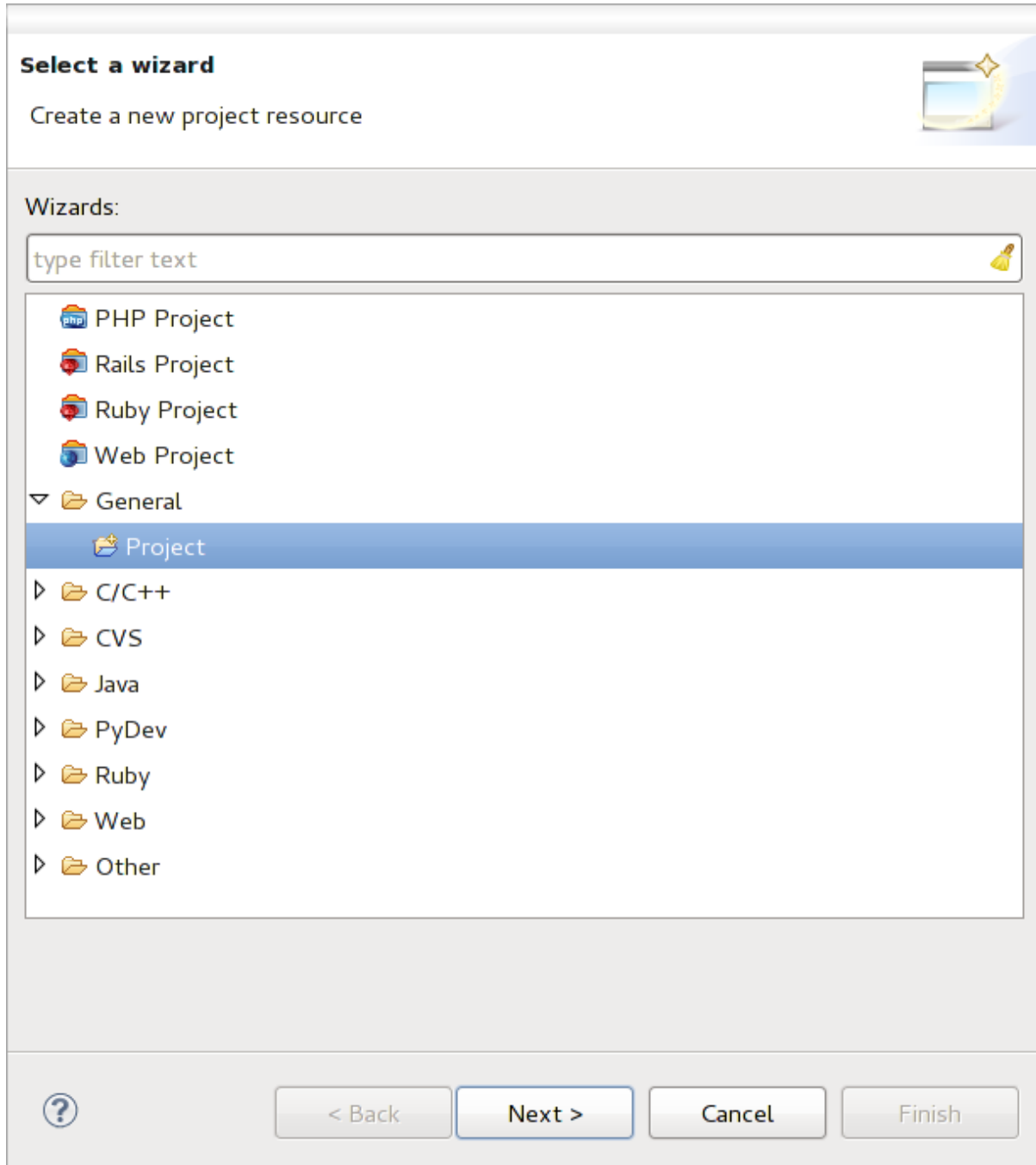


Figure 14.1: Proiectul Eclipse

14.2.4 Configurarea depanatorului

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

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

În acest moment, Eclipse așteaptă o conexiune de la QGIS către serverul de depanare, iar când QGIS se va conecta la serverul de depanare va fi permis controlul scripturilor Python. Exact pentru acest lucru am instalat plugin-ul *Remote Debug*. Deci, startați QGIS, în cazul în care nu ați făcut-o deja și efectuați clic pe simbolul insectei.

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

```

87         self.verticalExaggeration = 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 )

```

Figure 14.2: 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.

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

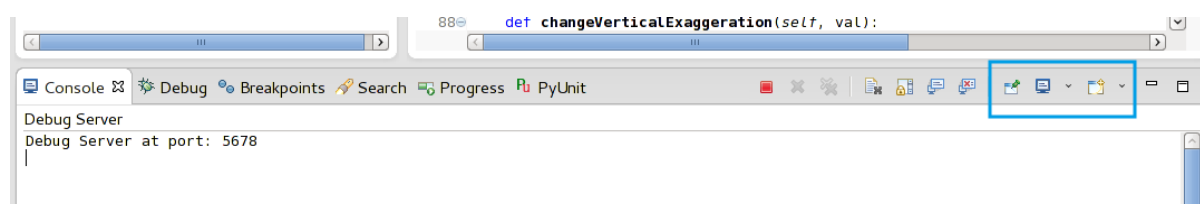


Figure 14.3: Consola de depanare PyDev

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

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

14.2.5 Configurați Eclipse pentru a înțelege API-ul

O caracteristică facilă este de a pregăti Eclipse pentru API-ul QGIS. Aceasta va permite verificarea eventualelor greșeli 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 fișierele bibliotecii QGIS și primește toate informațiile de acolo. Singurul lucru pe care trebuie să-l faceți este de a-i indica lui Eclipse unde să găsească bibliotecile.

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

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

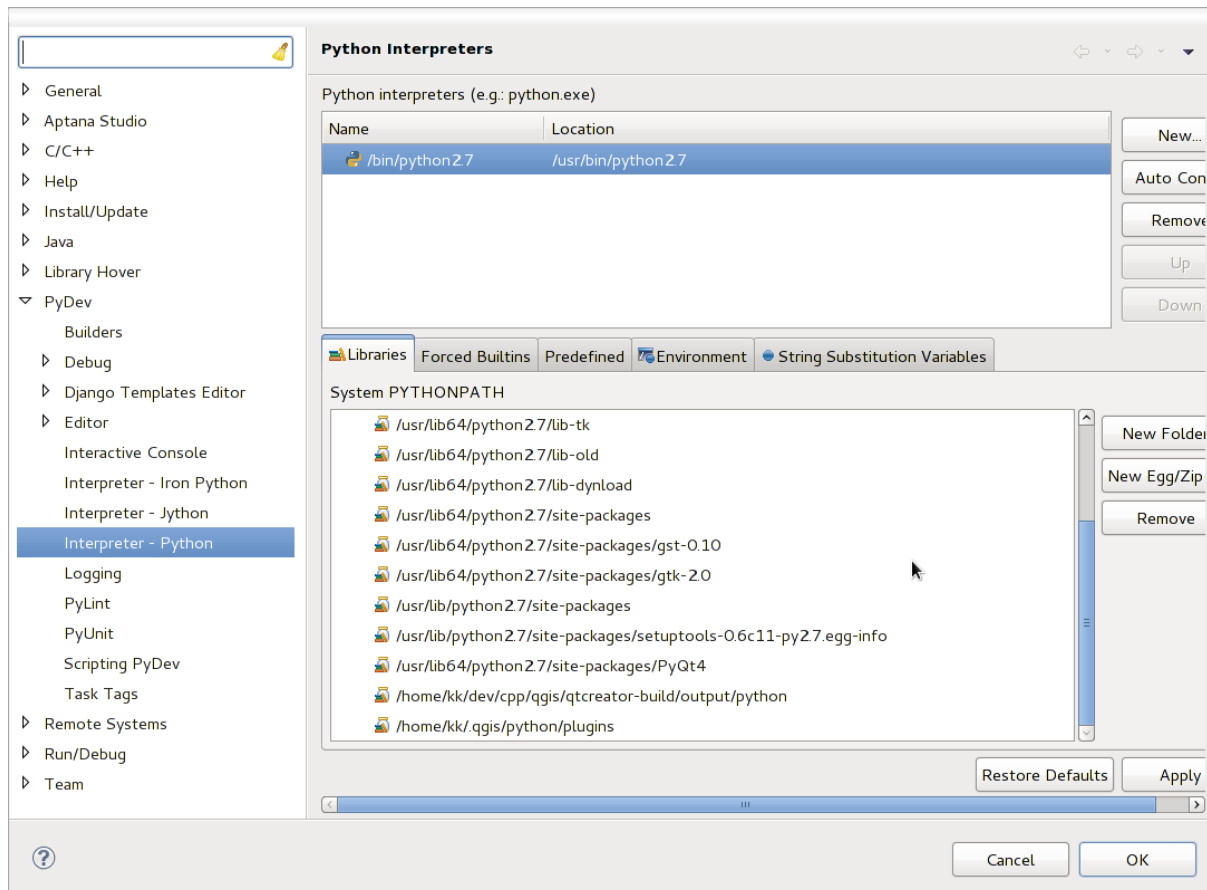


Figure 14.4: Consola de depanare PyDev

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

Ar trebui să adăugați, de asemenea, propriul director de plugin-uri aici (în Linux este `~/qgis2/python/plugins`).

Apoi deschideți tab-ul *Includeri Forțate*, faceți clic pe *Nou...* și introduceți `qgis`. Acest lucru va face ca Eclipse să analizeze API-ul QGIS. Probabil doriți, de asemenea, ca Eclipse să știe și despre API-ul PyQt4. Prin urmare, adăugați și `PyQt4` ca includere forțată. Probabil că ar trebui să se afle deja în fila bibliotecilor.

Faceți clic pe *OK* și ați terminat.

Note: Notă: la orice modificare a API-ului QGIS (de exemplu, în urma compilării QGIS master și a modificării

fișierului SIP), ar trebui să mergeți înapoi la această pagină și pur și simplu să faceți clic pe *Aplicare*. Acest lucru va face ca Eclipse să analizeze toate bibliotecile din nou.

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

14.3 Depanarea cu ajutorul PDB

Dacă nu folosiți un IDE, cum ar fi Eclipse, puteți depana folosind PDB, urmând acești pași.

Mai întâi, adăugați acest cod în locul în care doriți depanarea

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

Apoi executați QGIS din linia de comandă.

În Linux rulați:

```
$ ./Qgis
```

În Mac OS X rulați:

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

Iar când aplicația atinge punctul de întrerupere aveți posibilitatea de a tasta în consolă!

DE EFECTUAT: Adăugați informații pentru testare

Utilizarea straturilor plugin-ului

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

DE EFECTUAT: Verificați corectitudinea și elaborați cazuri de utilizare corectă pentru `QgsPluginLayer`, ...

15.1 Subclasarea `QgsPluginLayer`

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

```
class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

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

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

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

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

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

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

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

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

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

Compatibilitatea cu versiunile QGIS anterioare

16.1 Meniul plugin-ului

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

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

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

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

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

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

Lansarea plugin-ului dvs.

- Metadate și nume
- Codul și ajutorul
- Depozitul oficial al plugin-urilor python
 - Permisuni
 - Managementul încrederii
 - Validare
 - Structura plugin-ului

O dată ce plugin-ul este gata și credeți că el ar putea fi de ajutor pentru unii utilizatori, nu ezitați să-l încărcați la *Depozitul oficial al plugin-urilor python*. Pe acea pagină puteți găsi instrucțiuni de împachetare și de pregătire a plugin-ului, pentru a lucra bine cu programul de instalare. Sau, în cazul în care ați dori să înființați un depozit propriu pentru plugin-uri, creați un simplu fișier XML, care va lista plugin-urile și metadatele lor; de exemplu, consultați *depozitele pentru plugin-uri*.

Vă rugăm să acordați o grijă deosebită următoarelor recomandări:

17.1 Metadate și nume

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

17.2 Codul și ajutorul

- nu includeți fișierul generat (ui_*.py, resources_rc.py, fișiere de ajutor generate...) și chestii inutile (ex: .gitignore) în depozit

- adăugați pluginul în meniul corespunzător (Vector, Raster, Web, Bază de date)
- atunci când este cazul (plugin-uri efectuând analize), luați în considerare adăugarea plugin-ului ca subplugin al cadrului de Procesare: acest lucru va permite utilizatorilor să-l rulați în lot, să-l integrați în fluxurile de lucru mai complexe, eliberându-vă de povara proiectării unei interfețe
- includeți cel puțin documentația minimă și, dacă este util pentru testare și înțelegere, datele eșantion.

17.3 Depozitul oficial al plugin-urilor python

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

Pentru a folosi depozitul oficial, trebuie să obțineți un ID OSGEO din portalul web [OSGEO](#).

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

DE EFECTUAT: Introduceți un link către documentul guvernantei

17.3.1 Permisuni

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

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

17.3.2 Managementul încrederii

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

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

17.3.3 Validare

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

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

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

17.3.4 Structura plugin-ului

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

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

Fragmente de cod

- Cum să apelăm o metodă printr-o combinație rapidă de taste
- Inversarea Stării Straturilor
- Cum să accesați tabelul de atribute al entităților selectate

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

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

În plug-in adăugați în `initGui()`

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

În `unload()` adă

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

Metoda care este chemată atunci când se apasă tasta F7

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

18.2 Inversarea Stării Straturilor

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

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

18.3 Cum să accesați tabelul de atribute al entităților selectate

```
def changeValue(self, value):
    layer = self.iface.activeLayer()
    if(layer):
        nF = layer.selectedFeatureCount()
        if (nF > 0):
            layer.startEditing()
            ob = layer.selectedFeaturesIds()
            b = QVariant(value)
            if (nF > 1):
                for i in ob:
                    layer.changeAttributeValue(int(i), 1, b) # 1 being the second column
            else:
                layer.changeAttributeValue(int(ob[0]), 1, b) # 1 being the second column
            layer.commitChanges()
        else:
            QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select at least one feature")
    else:
        QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select a layer")
```

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

```
self.changeValue(50)
```

Scrierea unui plugin Processing

- Crearea unui plug-in care adaugă un furnizor de algoritm
- Crearea unui plug-in care conține un set de script-uri de procesare

În funcție de tipul de plugin avut în vedere, adăugarea funcționalității acestuia ca algoritm (sau ca set) pentru Processing, ar putea fi o opțiune mai bună. Acest lucru ar consta într-o mai bună integrare în cadrul QGIS, o funcționalitate suplimentară (din moment ce poate fi rulat din cadrul componentelor Processing, cum ar fi modelatorul sau interfața de prelucrare în serie), precum și un timp de dezvoltare mai rapid (atât timp cât Processing vă va scuti de o mare parte din muncă).

Acest document descrie modul de creare a un nou plug-in care se va regăsi ca algoritm în Processing.

Există două mecanisme principale pentru a face asta:

- Crearea unui plug-in care adaugă un furnizor de algoritm: Această opțiune este mult mai complexă, dar oferă mai multă flexibilitate
- Crearea unui plug-in care conține un set de script-uri de procesare: Cea mai simplă soluție, aveți nevoie doar de un set de fișiere cu cod specific pentru Processing.

19.1 Crearea unui plug-in care adaugă un furnizor de algoritm

Pentru a crea un furnizor pentru algoritm, urmați acești pași:

- Instalarea Plugin Builder
- Creați un plugin nou, utilizând Plugin Builder. În cazul în care Plugin Builder vă cere șablonul de utilizat, selectați “Furnizor Processing”.
- Plugin-ul creat conține un furnizor cu un singur algoritm. Atât fișierul furnizorului cât și cel al algoritmului sunt complet comentate și conțin informații cu privire la modul de modificare a furnizorului și de adăugare a algoritmilor suplimentari.

19.2 Crearea unui plug-in care conține un set de script-uri de procesare

Pentru a crea un set de script-uri de procesare, urmați acești pași:

- Creați script-urile așa cum s-a descris în Cartea Rețetelor PyQGIS. Toate script-urile pe care doriți să le adăugați, ar trebui să fie disponibile în caseta instrumentelor Processing.
- În grupul *Script-urilor/Instrumentelor* din caseta instrumentelor Processing, efectuați dublu-clic pe elementul de *Creare a colecției de script-uri a plugin-ului*. Veți vedea o fereastră în care ar trebui să selectați

script-urile ce se vor adăuga plugin-ului (din setul celor disponibile în caseta de instrumente), precum și unele informații suplimentare, necesare pentru metadatele plugin-ului.

- Faceți clic pe Ok, pentru a fi creat plugin-ul.
- Puteți adăuga script-uri suplimentare în plugin, prin adăugarea fișierelor cu script-uri python în dosarul *script*, din dosarul plugin-ului rezultat.

Biblioteca de analiză a rețelor

- Informații generale
- Construirea unui graf
- Analiza grafului
 - Găsirea celor mai scurte căi
 - Ariile de disponibilitate

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

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

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

20.1 Informații generale

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

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

20.2 Construirea unui graf

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

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

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

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

Convertorul din strat vectorial în graf este dezvoltat folosind modelul de programare al **Constructorului**. De construirea grafului răspunde așa-numitul Director. Există doar un singur Director pentru moment: `QgsLineVectorLayerDirector`. Directorul stabilește setările de bază ale Constructorului, care vor fi folosite pentru a construi

un graf dintr-un strat vectorial de tip linie. În prezent, ca și în cazul Directorului, există doar un singur Constructor: `QgsGraphBuilder`, care creează obiecte <http://qgis.org/api/classQgsGraph.html>>'_ . Este posibil să doriți implementarea propriilor constructori care să construiască grafuri compatibile cu bibliotecile, cum ar fi BGL sau NetworkX.

Pentru a calcula proprietățile marginii este utilizată [strategia modelului de programare](#). Pentru moment doar strategia `QgsDistanceArcProperter` este disponibilă, care ia în considerare lungimea traseului. Puteți implementa propria strategie, care va folosi toți parametrii necesari. De exemplu, plugin-ul RoadGraph folosește strategia care calculează timpul de călătorie, folosind lungimea muchiei și valoarea vitezei din atribute.

Este timpul de a aprofunda acest proces.

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

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

Apoi, câteva exemple pentru crearea unui director

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

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

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

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

Iată lista completă a ceea ce înseamnă acești parametri:

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

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

```
properter = QgsDistanceArcProperter()
```

Apoi spuneți directorului despre această strategie

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

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

- `crs` — sistemul de coordonate de referință de utilizat. Argument obligatoriu.
- `otfEnabled` — utilizați sau nu reproiectarea “din zbor”. În mod implicit `const:True` (folosiți OTF).
- `topologyTolerance` — toleranța topologică. Valoarea implicită este 0.
- `elipsoidID` — elipsoidul de utilizat. În mod implicit “WGS84”.

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

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

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

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

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

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

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

Cu următorul cod putem obține indecșii punctelor noastre

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

20.3 Analiza grafului

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

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

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

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

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

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

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

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

Metoda `dijkstra()` are aceleași argumente, dar întoarce două matrici. În prima matrice, elementul `i` conține indicele marginii de intrare, sau `-1` în cazul în care nu există margini de intrare. În a doua matrice, elementul `i` conține distanța de la rădăcina arborelui la nodul `i`, sau `DOUBLE_MAX` dacă vertexul nu poate fi accesat pornind de la rădăcină.

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

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

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

Același 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())

```

20.3.1 Găsirea celor mai scurte căi

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

```

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

```

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

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

```

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

p.append(tStart)

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

for pnt in p:
    rb.addPoint(pnt)

```

20.3.2 Ariile de disponibilitate

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

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

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

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

Iată un exemplu

```

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

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

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

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

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

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

idStart = graph.findVertex(tStart)

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

```

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

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

Plugin-uri Python pentru Serverul QGIS

- Arhitectura Plugin-urilor de Filtrare de pe Server
 - requestReady
 - sendResponse
 - responseComplete
- Tratarea excepțiilor provenite de la un plugin
- Scrierea unui plugin pentru server
 - Fișierele Plugin-ului
 - `__init__.py`
 - `HelloServer.py`
 - Modificarea intrării
 - Modificarea sau înlocuirea rezultatului
- Plugin-ul de control al accesului
 - Fișierele Plugin-ului
 - `__init__.py`
 - `AccessControl.py`
 - `layerFilterExpression`
 - `layerFilterSubsetString`
 - `layerPermissions`
 - `authorizedLayerAttributes`
 - `allowToEdit`
 - `cacheKey`

Plugin-urile Python pot rula și pe QGIS Server (a se vedea: *label_qgisserver*): prin utilizarea *interfeței server-ului* (`QgsServerInterface`), un plugin scris în Python, care rulează pe server, poate modifica comportamentul serviciilor de bază existente (**WMS**, **WFS** etc.).

Cu ajutorul *interfeței de filtrare a server-ului* (`QgsServerFilter`) putem schimba parametrii de intrare, rezultatul generat, sau putem furniza noi servicii.

Cu ajutorul *interfeței de control al accesului* (`QgsAccessControlFilter`) putem aplica unele restricții de acces asupra cererilor.

21.1 Arhitectura Plugin-urilor de Filtrare de pe Server

Plugin-urile Python de pe Server sunt încărcate o dată ce pornește aplicația FCGLI. Acestea înregistrează una sau mai multe clase `QgsServerFilter` (din acest punct, ar fi util să aruncați o privire rapidă la [Documentația API a plugin-urilor pentru server](#)). Fiecare filtru ar trebui să implementeze cel puțin una dintre cele trei funcții de reapelare:

- `requestReady()`
- `responseComplete()`

- `sendResponse ()`

Toate filtrele au acces la obiectul cerere/răspuns (`QgsRequestHandler`), putându-i manipula toate proprietățile (de intrare/ieșire) și tratându-i excepțiile (deși într-un mod cu totul particular, după cum vom vedea mai jos).

Mai jos se află un pseudocod care prezintă o sesiune tipică de server și reapelarea filtrelor:

- **se obține cererea de intrare**
 - se creează o rutină de tratare a cererilor GET/POST/SOAP
 - se transmite cererea către o instanță a clasei `QgsServerInterface`
 - se apelează filtrele `requestReady ()` ale plugin-urilor
 - **în cazul în care nu există un răspuns**
 - * **dacă SERVICE este de tipul WMS/WFS/WCS**
 - **se creează serverul WMS/WFS/WCS**
 - se apelează funcția `executeRequest ()` a serverului și, eventual, a funcției `sendResponse ()` a filtrelor plugin-ului, atunci când are loc generarea rezultatului, sau stocarea fluxului de ieșire cu octeți și a tipului conținutului din rutina de tratare a cererii
 - * se apelează filtrele `responseComplete ()` ale plugin-urilor
 - se apelează filtrele `sendResponse ()` ale plugin-urilor
 - request handler output the response

Următoarele paragrafe descriu, în detaliu, funcțiile de reapelare disponibile.

21.1.1 requestReady

Este apelată atunci când cererea este pregătită: adresa și datele primite au fost analizate și, înainte de a intra în comutatorul serviciilor de bază (WMS, WFS, etc), acesta este punctul în care se poate interveni asupra datelor de intrare, putându-se efectua acțiuni de genul:

- autentificare/autorizare
- redirectări
- adăugarea/eliminarea anumitor parametri (denumirile tipurilor, de exemplu)
- tratarea excepțiilor

Ați putea chiar să substituiți în întregime un serviciu de bază, prin schimbarea parametrului **SERVICE**, astfel, ocolindu-se complet serviciul de bază (deși, acest lucru nu ar avea prea mult sens).

21.1.2 sendResponse

Este apelată de fiecare dată când ieșirea este dirijată către **FCGI stdout** (și de acolo, înspre client), acest lucru făcându-se, în mod normal, după ce serviciile de bază și-au finalizat procesarea și după ce a fost apelată funcția de interceptare `responseComplete`; totuși, în anumite cazuri, fișierul XML poate deveni extrem de mare, de aceea a fost nevoie de implementarea unei funcțiuni de transfer continuu a fluxului XML (WFS GetFeature este una dintre ele), în acest caz, funcția `sendResponse ()` fiind apelată de mai multe ori înainte de încheierea răspunsului (și înainte de a fi apelată `responseComplete ()`). În consecință, `sendResponse ()` este, în mod normal, apelată doar o dată, însă, în mod excepțional, apelarea poate avea loc de mai multe ori, iar în acest caz (și numai în acest caz) va fi apelată înaintea funcției `responseComplete ()`.

`sendResponse ()` reprezintă cel mai bun loc pentru manipularea directă a rezultatului serviciilor de bază și, în timp ce `responseComplete ()` constă, de asemenea, într-o opțiune, `sendResponse ()` este singura opțiune viabilă în cazul serviciilor de redare continuă a fluxului.

21.1.3 responseComplete

Este apelată o singură dată, atunci când procesele serviciilor de bază (în cazul în care sunt implicate) s-au încheiat, iar cererea este gata de a fi transmisă clientului. Așa cum s-a discutat mai sus, este apelată, de obicei, înaintea funcției `sendResponse()`, cu excepția serviciilor de redare continuă a fluxului (sau a altor filtre ale plugin-urilor), care ar putea apela `sendResponse()` mai devreme.

`responseComplete()` reprezintă locul ideal pentru implementarea unor noi servicii (WPS sau servicii personalizate), precum și pentru a efectua intervenții directe asupra rezultatului care provine de la serviciile de bază (de exemplu, pentru a adăuga un filigran pe o imagine WMS).

21.2 Tratarea excepțiilor provenite de la un plugin

Mai sunt ceva acțiuni de efectuat în acest capitol: implementarea curentă poate distinge între excepțiile tratate și cele netratate, prin setarea proprietății `QgsRequestHandler` pentru o instanță a clasei `QgsMapServiceException`; în acest fel, codul C++ principal poate să intercepteze excepțiile Python tratate și să le ignore pe cele netratate (sau mai bine: să le jurnalizeze).

Această abordare funcționează în principiu, dar nu este în spiritul limbajului “python”: o abordare mai bună ar fi de a face vizibile excepțiile din codul python la nivelul buclei C++, pentru a fi manipulată acolo.

21.3 Scrierea unui plugin pentru server

Un server de plugin-uri reprezintă doar un plugin Python standard, pentru QGIS, așa cum este descris în *Dezvoltarea plugin-urilor Python*, care oferă o interfață suplimentară (sau alternativă): un plugin tipic pentru QGIS desktop are acces la aplicație prin instanța `QgisInterface`, pe când un plugin pentru server are acces, în plus, la clasa `QgsServerInterface`.

Pentru a spune Serverului QGIS că un plugin are o interfață de server, este necesară o intrare de metadate specială (în `metadata.txt`)

```
server=True
```

Exemplul de plugin discutat aici (cu mult mai multe exemple de filtre), este disponibil pe github: [QGIS HelloServer Example Plugin](#)

21.3.1 Fișierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin pentru server

```
PYTHON_PLUGINS_PATH/
HelloServer/
  __init__.py    --> *required*
  HelloServer.py --> *required*
  metadata.txt  --> *required*
```

21.3.2 __init__.py

Acest fișier este cerut de sistemul de import al Python. De asemenea, serverul QGIS necesită ca acest fișier să conțină o funcție `serverClassFactory()`, care este apelată atunci când plugin-ul se încarcă în QGIS Server, la pornirea serverului. Acesta primește referința către o instanță a `QgsServerInterface` și trebuie să returneze instanța clasei plugin-ului dvs. Iată cum arată fișierul `__init__.py` al exemplului de plugin:

```
# -*- coding: utf-8 -*-  
  
def serverClassFactory(serverIface):  
    from HelloServer import HelloServerServer  
    return HelloServerServer(serverIface)
```

21.3.3 HelloServer.py

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

Un plug-in de server este format, de obicei, dintr-una sau mai multe funcții Callback, ambalate în obiecte denumite `QgsServerFilter`.

Fiecare `QgsServerFilter` implementează una sau mai multe dintre următoarele funcții callback:

- `requestReady()`
- `responseComplete()`
- `sendResponse()`

Exemplul următor implementează un filtru minimal, care generează textul *HelloServer!* atunci când parametrul **SERVICE** este egal cu "HELLO":

```
from qgis.server import *  
from qgis.core import *  
  
class HelloFilter(QgsServerFilter):  
  
    def __init__(self, serverIface):  
        super(HelloFilter, self).__init__(serverIface)  
  
    def responseComplete(self):  
        request = self.serverInterface().requestHandler()  
        params = request.parameterMap()  
        if params.get('SERVICE', '').upper() == 'HELLO':  
            request.clearHeaders()  
            request.setHeader('Content-type', 'text/plain')  
            request.clearBody()  
            request.appendBody('HelloServer!')
```

Filtrele trebuie să fie înregistrate în **serverIface** ca în exemplul următor:

```
class HelloServerServer:  
    def __init__(self, serverIface):  
        # Save reference to the QGIS server interface  
        self.serverIface = serverIface  
        serverIface.registerFilter( HelloFilter, 100 )
```

Al doilea parametru al funcției `registerFilter()` permite stabilirea unei priorități, care definește ordinea pentru funcțiile callback cu același nume (prioritatea inferioară este invocată mai întâi).

Prin utilizarea celor trei funcții callback, plugin-urile pot manipula intrarea și/sau ieșirea serverului în mai multe moduri diferite. În orice moment, instanța plugin-ului are acces la `QgsRequestHandler` prin intermediul clasei `QgsServerInterface`. Clasa `QgsRequestHandler` dispune de o mulțime de metode care pot fi utilizate pentru modificarea parametrilor de intrare, înainte de a intra în nucleul de prelucrare al serverului (prin utilizarea `requestReady()`) sau în urma procesării cererii de către serviciile de bază (prin utilizarea `sendResponse()`).

Următorul exemplu demonstrează câteva cazuri de utilizare obișnuită:

21.3.4 Modificarea intrării

Exemplul de plugin conține un test care modifică parametrii de intrare provenind din șirul de interogare, în acest exemplu un nou parametru fiind injectat în *parameterMap* (deja analizat), parametrul fiind apoi vizibil tuturor serviciilor de bază (WMS etc.); la încheierea procesării, efectuate de către serviciile de bază, vom verifica dacă parametrul este încă acolo:

```
from qgis.server import *
from qgis.core import *

class ParamsFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(ParamsFilter, self).__init__(serverIface)

    def requestReady(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.responseComplete", 'plugin', QgsMess
        else:
            QgsMessageLog.logMessage("FAIL - ParamsFilter.responseComplete", 'plugin', QgsMess
```

Acesta este un extras a ceea ce vom vedea în fișierul jurnal:

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServe
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin He
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python pl
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&req
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair :
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms] inserting pair l
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter plugin default request
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.requ
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default configuration file path: /l
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking byte array is ok to
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, settin
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.respo
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - Paramsl
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFilt
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP response
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.sendl
```

În linia 13, șirul “SUCCESS” indică faptul că plugin-ul a trecut testul.

Aceeași tehnică poate fi exploatată pentru a utiliza un serviciu personalizat în locul unuia de bază: de exemplu, ați putea sări peste o cerere **WFS SERVICE**, sau peste oricare altă cerere de bază, doar prin schimbarea parametrului **SERVICE** în ceva diferit, iar serviciul de bază va fi omis; în acel caz, veți puteți injecta datele dvs. în interiorul rezultatului, trimițându-le clientului (acest lucru este explicat în continuare).

21.3.5 Modificarea sau înlocuirea rezultatului

Exemplul filtrului cu filigran vă arată cum să înlocuiți rezultatul WMS cu o nouă imagine, obținută prin adăugarea unei imagini filigran în partea de sus a imaginii WMS, generată de serviciul de bază WMS:

```
import os

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

class WatermarkFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(WatermarkFilter, self).__init__(serverIface)

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        # Do some checks
        if (request.parameter('SERVICE').upper() == 'WMS' \
            and request.parameter('REQUEST').upper() == 'GETMAP' \
            and not request.exceptionRaised()):
            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image ready %s" % request
                # Get the image
                img = QImage()
                img.loadFromData(request.body())
                # Adds the watermark
                watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
                p = QPainter(img)
                p.drawImage(QRect( 20, 20, 40, 40), watermark)
                p.end()
                ba = QByteArray()
                buffer = QBuffer(ba)
                buffer.open(QIODevice.WriteOnly)
                img.save(buffer, "PNG")
                # Set the body
                request.clearBody()
                request.appendBody(ba)
```

În cadrul acestui exemplu, este verificată valoarea parametrului **SERVICE**, iar în cazul în care cererea de intrare este un **WMS GETMAP** și nici un fel de excepții nu au fost stabilite de către un plugin executat anterior, sau de către serviciul de bază (WMS în acest caz), imaginea generată de către WMS este preluată din zona tampon de ieșire, adăugându-i-se imaginea filigran. Pasul final este de a goli tamponul de ieșire și de-l înlocui cu imaginea nou generată. Rețineți că într-o situație reală, ar trebui, de asemenea, să verificați tipul imaginii solicitate în loc de a returna, în toate cazurile, PNG-ul.

21.4 Plugin-ul de control al accesului

21.4.1 Fișierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin pentru server:

```
PYTHON_PLUGINS_PATH/
  MyAccessControl/
    __init__.py    --> *required*
    AccessControl.py --> *required*
    metadata.txt  --> *required*
```

21.4.2 `__init__.py`

Acest fișier este cerut de sistemul de import al Python. Similar tuturor plugin-urilor pentru serverul QGIS, acest fișier trebuie să conțină o funcție `serverClassFactory()`, care este apelată atunci când plugin-ul se încarcă în QGIS Server, la pornirea serverului. Acesta primește referința către o instanță a `QgsServerInterface` și trebuie să returneze instanța clasei plugin-ului dvs. Iată cum arată fișierul `__init__.py` al exemplului de plugin:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from MyAccessControl.AccessControl import AccessControl
    return AccessControl(serverIface)
```

21.4.3 `AccessControl.py`

```
class AccessControl(QgsAccessControlFilter):

    def __init__(self, server_iface):
        super(QgsAccessControlFilter, self).__init__(server_iface)

    def layerFilterExpression(self, layer):
        """ Return an additional expression filter """
        return super(QgsAccessControlFilter, self).layerFilterExpression(layer)

    def layerFilterSubsetString(self, layer):
        """ Return an additional subset string (typically SQL) filter """
        return super(QgsAccessControlFilter, self).layerFilterSubsetString(layer)

    def layerPermissions(self, layer):
        """ Return the layer rights """
        return super(QgsAccessControlFilter, self).layerPermissions(layer)

    def authorizedLayerAttributes(self, layer, attributes):
        """ Return the authorised layer attributes """
        return super(QgsAccessControlFilter, self).authorizedLayerAttributes(layer, attributes)

    def allowToEdit(self, layer, feature):
        """ Are we authorise to modify the following geometry """
        return super(QgsAccessControlFilter, self).allowToEdit(layer, feature)

    def cacheKey(self):
        return super(QgsAccessControlFilter, self).cacheKey()
```

Acest exemplu oferă un acces deplin pentru oricine.

Este de datoria plugin-ului să știe cine este conectat.

Toate aceste metode au ca argument stratul, pentru a putea personaliza restricțiile pentru fiecare strat.

21.4.4 `layerFilterExpression`

Se folosește pentru a adăuga o Expresie de limitare a rezultatelor, ex.:

```
def layerFilterExpression(self, layer):
    return "$role = 'user'"
```

Pentru restrângerea la entitățile pentru care atributul “rol” are valoarea “user”.

21.4.5 layerFilterSubsetString

La fel ca și precedenta, dar folosește `SubsetString` (execută în baza de date)

```
def layerFilterSubsetString(self, layer):  
    return "role = 'user'"
```

Pentru restrângerea la entitățile pentru care atributul “rol” are valoarea “user”.

21.4.6 layerPermissions

Limitează accesul la strat.

Returnează un obiect de tip `QgsAccessControlFilter.LayerPermissions`, care are proprietățile:

- `canRead` pentru a-l vedea în `GetCapabilities` și pentru a avea acces de citire.
- `canInsert` pentru a putea insera o nouă entitate.
- `canUpdate` pentru a putea actualiza o entitate.
- `candelelete` pentru a fi putea șterge o entitate.

Exemplu:

```
def layerPermissions(self, layer):  
    rights = QgsAccessControlFilter.LayerPermissions()  
    rights.canRead = True  
    rights.canRead = rights.canInsert = rights.canUpdate = rights.canDelete = False  
    return rights
```

Pentru a permite tuturor accesul numai pentru citire.

21.4.7 authorizedLayerAttributes

Folosit pentru a reduce vizibilitatea unui subset specific de atribute.

Atributul argument returnează setul actual de atribute vizibile.

Exemplu:

```
def authorizedLayerAttributes(self, layer, attributes):  
    return [a for a in attributes if a != "role"]
```

Pentru a ascunde atributul ‘role’.

21.4.8 allowToEdit

Se folosește pentru a limita editarea unui subset specific de entități.

Este folosit în protocolul WFS-Transaction.

Exemplu:

```
def allowToEdit(self, layer, feature):  
    return feature.attribute('role') == 'user'
```

Pentru a putea modifica numai entitatea pentru care atributul “rol” are valoarea de “utilizator”.

21.4.9 cacheKey

Serverul QGIS menține o memorie tampon a capabilităților, de aceea, pentru a avea o memorie cache pentru fiecare rol, puteți specifica rolul cu ajutorul acestei metode. Sau puteți seta valoarea `None`, pentru a dezactiva complet memoria tampon.

- în execuție
 - aplicații personalizate, 4
- încărcare
 - Fișiere GPX, 10
 - Geometrii MySQL, 10
 - proiecte, 7
 - straturi cu text delimitat, 10
 - Straturi OGR, 9
 - Straturi PostGIS, 9
 - straturi raster, 10
 - Straturi SpatiaLite, 10
 - straturi vectoriale, 9
 - Straturi WMS, 11
- API, 1
- aplicații personalizate
 - în execuție, 4
- aplicații personalizate; script-uri de sine stătătoare
 - Python, 3
- atribute
 - straturi vectoriale entități, 17
- calcularea valorilor, 50
- consolă
 - Python, 2
- entități
 - atribute, straturi vectoriale, 17
 - straturi vectoriale iterarea, 18
 - straturi vectoriale selecție, 17
- expresii, 50
 - evaluare, 52
 - parsare, 52
- Fișiere GPX
 - încărcare, 10
- filtrare, 50
- furnizor de memorie, 24
- geometrie
 - accesare, 35
 - construire, 35
 - manipulare, 33
 - prediccate și operațiuni, 36
- Geometrii MySQL
 - încărcare, 10
- ieșire
 - folosirea Compozitorului de Hărți, 48
 - imagine raster, 49
 - PDF, 50
- index spatial
 - folosind, 22
- inițializare
 - Python, 1
- interogare
 - straturi raster, 15
- iterarea
 - entități, straturi vectoriale, 18
- mediu
 - PYQGIS_STARTUP, 1
- metadata, 64
- metadata.txt, 64, 97
- metadata, 97
- personalizat
 - rendere, 31
- plugin-uri, 79
 - apelarea unei metode printr-o combinație rapidă de taste, 83
 - atributele de acces ale entităților selectate, 83
 - comutarea straturilor, 83
 - depozitul oficial al plugin-urilor python, 80
 - dezvoltare, 59
 - documentație, 66
 - fișier de resurse, 66
 - fragmente de cod, 67
 - implementare help, 66
 - lansarea, 74
 - metadata.txt, 62, 64, 97
 - scriere, 62
 - scriere cod, 62
 - testare, 74
- plugin-uri pentru server
 - dezvoltare, 94
 - metadata.txt, 97
- proiecții, 40
- proiecte
 - încărcare, 7
- PYQGIS_STARTUP
 - mediu, 1

- Python
 - aplicații personalizate; script-uri de sine stătătoare, 3
 - consolă, 2
 - dezvoltarea plugin-urilor, 59
 - dezvoltarea plugin-urilor pentru server, 94
 - inițializare, 1
 - plugin-uri, 2
 - startup.py, 2
- randare hartă, 46
 - simplicu, 47
- rastere
 - multibandă, 14
 - simplică bandă, 14
- recitire
 - straturi raster, 15
- registru straturilor de hartă, 11
 - adăugarea unui strat, 11
- render cu simbol gradual, 27
- render cu simbologie clasificată, 27
- render cu un singur simbol , 26
- rendere
 - personalizat, 31
- resources.qrc, 66
- selecție
 - entități, straturi vectoriale, 17
- setări
 - citire, 53
 - global, 55
 - proiect, 55
 - stocare, 53
 - strat de hartă, 56
- simbologie
 - render cu simbol clasificat, 27
 - render cu simbol gradual, 27
 - render cu un singur simbol , 26
 - vechi, 33
- simboluri
 - lucrul cu, 28
- sisteme de coordonate de referință, 39
- startup.py
 - Python, 2
- straturi cu text delimitat
 - încărcare, 10
- Straturi OGR
 - încărcare, 9
- Straturi PostGIS
 - încărcare, 9
- straturi raster
 - încărcare, 10
 - detalii, 13
 - folosind, 12
 - interogare, 15
 - recitire, 15
 - render, 13
- Straturi Spatialite
 - încărcare, 10
- straturi vectoriale
 - încărcare, 9
 - editare, 20
 - entități atribute, 17
 - iterarea entității, 18
 - scris, 23
 - selecție entități, 17
 - simbologie, 25
- Straturi WMS
 - încărcare, 11
- straturile plugin-ului, 74
 - subclasarea QgsPluginLayer, 75
- straturile simbolului
 - crearea tipurilor personalizate, 29
 - lucrul cu, 29
- suportul hărții, 40
 - încapsulare, 41
 - arhitectură, 41
 - benzi de cauciuc, 43
 - dezvoltarea elementelor personalizate pentru suportul de hartă , 45
 - dezvoltarea instrumentelor de hartă personalizate, 44
 - instrumente pentru hartă, 42
 - marcaje vertex, 43
- tipărire hartă, 46