PyQGIS developer cookbook

Release 2.18

QGIS Project

08. April 2019

Contents

1 Introductie **1**
- 1.1 Run Python code when QGIS starts . . . 1
- 1.2 Python Console . . . 2
- 1.3 Plug-ins in Python . . . 3
- 1.4 Toepassingen in Python . . . 3

2 Projecten laden **7**

3 Lagen laden **9**
- 3.1 Vectorlagen . . . 9
- 3.2 Rasterlagen . . . 11
- 3.3 Map Layer Registry . . . 11

4 Rasterlagen gebruiken **13**
- 4.1 Details laag . . . 13
- 4.2 Renderer . . . 13
- 4.3 Refreshing Layers . . . 15
- 4.4 Waarden bevragen . . . 15

5 Vectorlagen gebruiken **17**
- 5.1 Informatie over attributen ophalen . . . 17
- 5.2 Objecten selecteren . . . 18
- 5.3 Itereren over vectorlagen . . . 18
- 5.4 Vectorlagen bewerken . . . 20
- 5.5 Vectorlagen bewerken met een bewerkingsbuffer . . . 22
- 5.6 Ruimtelijke index gebruiken . . . 23
- 5.7 Writing Vector Layers . . . 23
- 5.8 Memory Provider . . . 24
- 5.9 Uiterlijk (symbologie) van vectorlagen . . . 26
- 5.10 Meer onderwerpen . . . 33

6 Afhandeling van geometrie **35**
- 6.1 Construeren van geometrie . . . 35
- 6.2 Toegang tot geometrie . . . 36
- 6.3 Predicaten en bewerking voor geometrieën . . . 36

7 Ondersteuning van projecties **39**
- 7.1 Coördinaten ReferentieSystemen . . . 39
- 7.2 Projections . . . 40

8 Using Map Canvas **41**
- 8.1 Kaartvenster inbedden . . . 41
- 8.2 Gereedschappen voor de kaart gebruiken in het kaartvenster . . . 42

8.3 Elastieken banden en markeringen voor punten . . . 43
8.4 Aangepaste gereedschappen voor de kaart schrijven . . . 44
8.5 Aangepaste items voor het kaartvenster schrijven . . . 45

9 Kaart renderen en afdrukken 47
9.1 Eenvoudig renderen . . . 47
9.2 Lagen met een verschillend CRS renderen . . . 48
9.3 Output using Map Composer . . . 48

10 Expressies, filteren en waarden berekenen 51
10.1 Parsen van expressies . . . 52
10.2 Evalueren van expressies . . . 52
10.3 Voorbeelden . . . 53

11 Instellingen lezen en opslaan 55

12 Communiceren met de gebruiker 57
12.1 Showing messages. The `QgsMessageBar` class . . . 57
12.2 Voortgang weergeven . . . 58
12.3 Loggen . . . 59

13 Python plug-ins ontwikkelen 61
13.1 Een plug-in schrijven . . . 62
13.2 Inhoud van de plug-in . . . 63
13.3 Documentatie . . . 67
13.4 Vertaling . . . 67

14 Infrastructuur voor authenticatie 71
14.1 Introductie . . . 71
14.2 Woordenlijst . . . 71
14.3 QgsAuthManager is het toegangspunt . . . 72
14.4 Plug-ins aanpassen om de infrastructuur voor authenticatie te gebruiken . . . 75
14.5 GUI's voor authenticatie . . . 75

15 Instellingen voor de IDE voor het schrijven en debuggen van plug-ins 79
15.1 Een opmerking voor het configureren van uw IDE op Windows . . . 79
15.2 Debuggen met behulp van Eclipse en PyDev . . . 80
15.3 Debuggen met behulp van PDB . . . 84

16 Plug-in-lagen gebruiken 85
16.1 Sub-klassen in QgsPluginLayer . . . 85

17 Compatibiliteit met oudere versies van QGIS 87
17.1 Menu Plug-ins . . . 87

18 Uw plug-in uitgeven 89
18.1 Metadata en namen . . . 89
18.2 Code en hulp . . . 90
18.3 Officiële Python plug-in opslagplaats . . . 90

19 Codesnippers 93
19.1 Hoe een methode aan te roepen met een sneltoets . . . 93
19.2 Hoe te schakelen tussen lagen . . . 93
19.3 Hoe toegang te krijgen tot de attributentabel van geselecteerde objecten . . . 94

20 Een plug-in voor Processing schrijven 95
20.1 Creating a plugin that adds an algorithm provider . . . 95
20.2 Creating a plugin that contains a set of processing scripts . . . 95

21 Bibliotheek Netwerkanalyse **97**
21.1 Algemene informatie 97
21.2 Een grafiek bouwen 97
21.3 Grafiekanalyse 99

22 Python plug-ins voor QGIS server **105**
22.1 Server Filter Plugins architecture 105
22.2 Een uitzondering opwerpen vanuit een plug-in 107
22.3 Een plug-in voor de server schrijven 107
22.4 Plug-in Access control 110

Index **115**

CHAPTER 1

Introductie

- Run Python code when QGIS starts
  - Omgevingsvariabele PYQGIS_STARTUP
  - Het bestand `startup.py`
- Python Console
- Plug-ins in Python
- Toepassingen in Python
  - PyQGIS gebruiken in zelfstandige scripts
  - PyQGIS gebruiken in aangepaste toepassing
  - Aangepaste toepassingen uitvoeren

This document is intended to work both as a tutorial and a reference guide. While it does not list all possible use cases, it should give a good overview of the principal functionality.

Starting from 0.9 release, QGIS has optional scripting support using Python language. We've decided for Python as it's one of the most favourite languages for scripting. PyQGIS bindings depend on SIP and PyQt4. The reason for using SIP instead of more widely used SWIG is that the whole QGIS code depends on Qt libraries. Python bindings for Qt (PyQt) are done also using SIP and this allows seamless integration of PyQGIS with PyQt.

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

- automatically run Python code when QGIS starts
- issue commands in Python console within QGIS
- create and use plugins in Python
- create custom applications based on QGIS API

Python bindings are also available for QGIS Server:

- starting from 2.8 release, Python plugins are also available on QGIS Server (see *Server Python Plugins*)
- starting from 2.11 version (Master at 2015-08-11), QGIS Server library has Python bindings that can be used to embed QGIS Server into a Python application.

There is a complete QGIS API reference that documents the classes from the QGIS libraries. Pythonic QGIS API is nearly identical to the API in C++.

A good resource when dealing with plugins is to download some plugins from plugin repository and examine their code. Also, the `python/plugins/` folder in your QGIS installation contains some plugin that you can use to learn how to develop such plugin and how to perform some of the most common tasks.

1.1 Run Python code when QGIS starts

Er zijn twee afzonderlijke methoden om Python-code uit te voeren elke keer als QGIS start.

1.1.1 Omgevingsvariabele PYQGIS_STARTUP

You can run Python code just before QGIS initialization completes by setting the `PYQGIS_STARTUP` environment variable to the path of an existing Python file.

This method is something you will probably rarely need, but worth mentioning here because it is one of the several ways to run Python code within QGIS and because this code will run before QGIS initialization is complete. This method is very useful for cleaning sys.path, which may have undesireable paths, or for isolating/loading the initial environ without requiring a virt env, e.g. homebrew or MacPorts installs on Mac.

1.1.2 Het bestand `startup.py`

Every time QGIS starts, the user's Python home directory (usually: `.qgis2/python`) is searched for a file named `startup.py`, if that file exists, it is executed by the embedded Python interpreter.

1.2 Python Console

For scripting, it is possible to take advantage of integrated Python console. It can be opened from menu: *Plugins* → *Python Console*. The console opens as a non-modal utility window:

Figure 1.1: QGIS Python-console

The screenshot above illustrates how to get the layer currently selected in the layer list, show its ID and optionally, if it is a vector layer, show the feature count. For interaction with QGIS environment, there is a `iface` variable, which is an instance of `QgsInterface`. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application.

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

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

For those which use the console often, it may be useful to set a shortcut for triggering the console (within menu *Settings* → *Configure shortcuts...*)

1.3 Plug-ins in Python

QGIS allows enhancement of its functionality using plugins. This was originally possible only with C++ language. With the addition of Python support to QGIS, it is also possible to use plugins written in Python. The main advantage over C++ plugins is its simplicity of distribution (no compiling for each platform needed) and easier development.

Many plugins covering various functionality have been written since the introduction of Python support. The plugin installer allows users to easily fetch, upgrade and remove Python plugins. See the Python Plugin Repositories page for various sources of plugins.

Plug-ins maken in Python is simpel, zie *Python plug-ins ontwikkelen* voor gedetailleerde instructies.

Notitie: Python plugins are also available in QGIS server (*label_qgisserver*), see *Python plug-ins voor QGIS server* for further details.

1.4 Toepassingen in Python

Often when processing some GIS data, it is handy to create some scripts for automating the process instead of doing the same task again and again. With PyQGIS, this is perfectly possible — import the `qgis.core` module, initialize it and you are ready for the processing.

Or you may want to create an interactive application that uses some GIS functionality — measure some data, export a map in PDF or any other functionality. The `qgis.gui` module additionally brings various GUI components, most notably the map canvas widget that can be very easily incorporated into the application with support for zooming, panning and/or any further custom map tools.

PyQGIS custom applications or standalone scripts must be configured to locate the QGIS resources such as projection information, providers for reading vector and raster layers, etc. QGIS Resources are initialized by adding a few lines to the beginning of your application or script. The code to initialize QGIS for custom applications and standalone scripts is similar, but examples of each are provided below.

Note: do *not* use `qgis.py` as a name for your test script — Python will not be able to import the bindings as the script's name will shadow them.

1.4.1 PyQGIS gebruiken in zelfstandige scripts

Initialiseer, om een zelfstandig script te starten, de bronnen voor QGIS aan het begin van het script, soortgelijk aan de volgende code:

```
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()
```

We beginnen door de module `qgis.core` te importeren en dan het pad voor het voorvoegsel te configureren. Het pad voor het voorvoegsel is de locatie waar QGIS is geïnstalleerd op uw systeem. Het wordt in het script geconfigureerd door de methode `setPrefixPath` aan te roepen. Het tweede argument van `setPrefixPath` is om de `True` in te stellen, die beheert of de standaard paden worden gebruikt.

Het pad voor de installatie van QGIS varieert per platform; de eenvoudigste manier om het voor uw systeem te vinden is door de *Python Console* te gebruiken vanuit QGIS en te kijken naar de uitvoer bij het uitvoeren van `QgsApplication.prefixPath()`.

Nadat het pad voor het voorvoegsel is geconfigureerd slaan we een verwijzing naar `QgsApplication` op in de variabele `qgs`. Het tweede argument wordt ingesteld op `False`, wat aangeeft dat we niet van plan zijn om de GUI te gebruiken omdat we een zelfstandig script schrijven. Met `QgsApplication` geconfigureerd laden we de gegevensproviders en registratie van lagen voor QGIS door de methode `qgs.initQgis()` aan te roepen. Met QGIS geïnitialiseerd zijn we klaar om de rest van het script te schrijven. Tenslotte sluiten we af door `qgs.exitQgis()` aan te roepen om de gegevensproviders en registratie van lagen uit het geheugen te verwijderen.

1.4.2 PyQGIS gebruiken in aangepaste toepassing

Het enige verschil tussen *PyQGIS gebruiken in zelfstandige scripts* en een aangepaste toepassing van PyQGIS is het tweede argument bij het instantiëren van `QgsApplication`. Geef op `True` in plaats van `False` om aan te geven dat we van plan zijn om een GUI te gaan gebruiken.

```
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()
```

Now you can work with QGIS API — load layers and do some processing or fire up a GUI with a map canvas. The possibilities are endless :-)

1.4.3 Aangepaste toepassingen uitvoeren

U zult uw systeem moeten vertellen waar te zoeken naar de bibliotheken van QGIS en de toepasselijke modules voor Python als zij nog niet op ene bekende locatie staan — anders zal Python gana klagen:

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

This can be fixed by setting the `PYTHONPATH` environment variable. In the following commands, `qgispath` should be replaced with your actual QGIS installation path:

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

The path to the PyQGIS modules is now known, however they depend on `qgis_core` and `qgis_gui` libraries (the Python modules serve only as wrappers). Path to these libraries is typically unknown for the operating system, so you get an import error again (the message might vary depending on the system):

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

Los dit op door de mappen waar de bibliotheken van QGIS zijn opgeslagen toe te voegen aan het zoekpad van de dynamische linker:

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

Deze opdrachten kunnen worden geplaatst in een bootstrap-script dat het opstarten voor zijn rekening zal nemen. Bij het uitrollen van toepaste toepassingen met behulp van PyQGIS, zijn er gewoonlijk twee mogelijkheden:

- require user to install QGIS on his platform prior to installing your application. The application installer should look for default locations of QGIS libraries and allow user to set the path if not found. This approach has the advantage of being simpler, however it requires user to do more steps.
- verpak QGIS tezamen met uw toepassing. Uitgeven van de toepassing ka uitdagender zijn en het pakket zal groter zijn, maar de gebruiker zal verlost zijn van de last van het downloaden en installeren van aanvullende stukken software.

De twee modellen van uitrollen kunnen worden gemixt - rol een zelfstandige toepassing uit op Windows en MacOS, laat voor Linux de installatie van QGIS over aan de gebruiker en diens pakketbeheer.

CHAPTER 2

Projecten laden

Sometimes you need to load an existing project from a plugin or (more often) when developing a stand-alone QGIS Python application (see: *Toepassingen in Python*).

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

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

In case you need to make some modifications to the project (for example add or remove some layers) and save your changes, you can call the `write()` method of your project instance. The `write()` method also accepts an optional `QFileInfo` that allows you to specify a path where the project will be saved:

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

Both `read()` and `write()` funtions return a boolean value that you can use to check if the operation was successful.

Notitie: If you are writing a QGIS standalone application, in order to synchronise the loaded project with the canvas you need to instanciate a `QgsLayerTreeMapCanvasBridge` as in the example below:

```
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'))
```

CHAPTER 3

Lagen laden

- Vectorlagen
- Rasterlagen
- Map Layer Registry

Laten we enkele lagen met gegevens openen. QGIS herkent vector- en rasterlagen. Aanvullend zijn anagepaste typen lagen beschikbaar, maar die zullen we hier niet bespreken.

3.1 Vectorlagen

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

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

De identificatie van de gegevensbron van de laag is een string en is specifiek voor elke vector gegevensprovider. De naam van de laag wordt gebruikt in de widget Lagenlijst. Het is belangrijk om te controleren of de laag met succes is geladen. Als dat niet zo was wordt een ongeldige instance van de laag teruggegeven.

The quickest way to open and display a vector layer in QGIS is the addVectorLayer function of the `QgisInterface`:

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

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step. The function returns the layer instance or *None* if the layer couldn't be loaded.

De volgende lijst geeft weer hoe toegang wordt verkregen tot verscheidene gegevensbronnen met behulp van vector gegevensproviders:

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

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

  - for dxf (note the internal options in data source uri):

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

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

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

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

Notitie: Het argument `False`, doorgegeven aan `uri.uri(False)`, voorkomt het uitbreiden van de parameters voor de configuratie voor authenticatie, indien u geen configuratie voor authenticatie gebruikt maakt dit argument geen enkel verschil.

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

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

Notitie: De string voor de provider is gestructureerd als een URL, dus moet het pad worden voorafgegaan door *file://*. Ook staat het als WKT (well known text) opgemaakte geometrieën toe als een alternatief voor velden "X" en "Y", en staat het toe dat het coördinaten referentiesysteem wordt gespecificeerd. Bijvoorbeeld

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

- GPX-bestanden — de "GPX"-gegevensprovider leest tracks, routes en waypoints uit GPX-bestanden. Het type (track/route/waypoint) moet worden gespecificeerd als deel van de URL om een bestand te openen:

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer name you like", "gpx")
```

- SpatiaLite database — Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier:

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

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

- MySQL op WKB gebaseerde geometrieën, via OGR — gegevensbron is de string voor de verbinding naar de tabel:

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

- WFS-verbinding:. de verbinding wordt gedefinieerd met een URI en het gebruiken van de provider `WFS`:

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

De URI kan worden gemaakt met behulp van de standaard bibliotheek `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))
```

Notitie: You can change the data source of an existing layer by calling `setDataSource()` on a `QgsVectorLayer` instance, as in the following example:

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

3.2 Rasterlagen

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

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

Similarly to vector layers, raster layers can be loaded using the addRasterLayer function of the `QgisInterface`:

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

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step.

Rasterlagen kunnen ook worden gemaakt vanuit een service voor WCS:

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

detailed URI settings can be found in provider documentation

Als alternatief kunt u een rasterlaag laden vanaf een server voor WMS. Momenteel is het echter niet mogelijk om toegang te krijgen tot het antwoord van GetCapabilities van de API — u moet weten welke lagen u wilt:

```
urlWithParams = 'url=http://irs.gis-lab.info/?layers=landsat&styles=&format=image/jpeg&crs=EPSG:4
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print "Layer failed to load!"
```

3.3 Map Layer Registry

If you would like to use the opened layers for rendering, do not forget to add them to map layer registry. The map layer registry takes ownership of layers and they can be later accessed from any part of the application by their unique ID. When the layer is removed from map layer registry, it gets deleted, too.

Adding a layer to the registry:

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

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

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

For a list of loaded layers and layer ids, use:

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

CHAPTER 4

Rasterlagen gebruiken

- Details laag
- Renderer
  - Enkelbands rasters
  - Multiband rasters
- Refreshing Layers
- Waarden bevragen

This sections lists various operations you can do with raster layers.

4.1 Details laag

A raster layer consists of one or more raster bands — it is referred to as either single band or multi band raster. One band represents a matrix of values. Usual color image (e.g. aerial photo) is a raster consisting of red, blue and green band. Single band layers typically represent either continuous variables (e.g. elevation) or discrete variables (e.g. land use). In some cases, a raster layer comes with a palette and raster values refer to colors stored in the palette:

```
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 Renderer

Wanneer een raster wordt geladen krijgt het een standaard renderer om te tekenen, gebaseerd op zijn type. Die kan worden gewijzigd, ofwel in de eigenschappen van de laag of programmatisch.

To query the current renderer:

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

To set a renderer use `setRenderer()` method of `QgsRasterLayer`. There are several available renderer classes (derived from `QgsRasterRenderer`):

- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

Enkelbands rasterlagen kunnen worden getekend ofwel in grijze kleuren (lage waarden = zwart, hoge waarden = wit) of met een algoritme voor pseudokleur dat kleuren toewijst voor de waarden uit de enkele band. Enkelbands rasters met een palet kunnen aanvullend worden getekend met behulp van hun palet. Multiband-lagen worden gewoonlijk getekend door de banden in kaart te brengen als RGB-kleuren. Een andere mogelijkheid is om slechts één band voor grijs of teken in pseudokleur te gebruiken.

The following sections explain how to query and modify the layer drawing style. After doing the changes, you might want to force update of map canvas, see *Refreshing Layers*.

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

4.2.1 Enkelbands rasters

Let's say we want to render our raster layer (assuming one band only) with colors ranging from green to yellow (for pixel values from 0 to 255). In the first stage we will prepare `QgsRasterShader` object and configure its shader function:

```
>>> 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)
```

De shader plaats de kleuren op de kaart zoals ze zijn gespecificeerd door zijn kleurenkaart. De kleurenkaart wordt verschaft als een lijst met items met pixelwaarde en de daarmee geassocieerde kleur. Er zijn drie modi voor de interpolatie van waarden:

- linear (`INTERPOLATED`): resulting color is linearly interpolated from the color map entries above and below the actual pixel value
- discrete (`DISCRETE`): color is used from the color map entry with equal or higher value
- exact (`EXACT`): color is not interpolated, only the pixels with value equal to color map entries are drawn

In the second step we will associate this shader with the raster layer:

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

The number 1 in the code above is band number (raster bands are indexed from one).

4.2.2 Multiband rasters

By default, QGIS maps the first three bands to red, green and blue values to create a color image (this is the `MultiBandColor` drawing style. In some cases you might want to override these setting. The following code interchanges red band (1) and green band (2):

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

In case only one band is necessary for visualization of the raster, single band drawing can be chosen — either gray levels or pseudocolor.

4.3 Refreshing Layers

If you do change layer symbology and would like ensure that the changes are immediately visible to the user, call these methods

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

The first call will ensure that the cached image of rendered layer is erased in case render caching is turned on. This functionality is available from QGIS 1.4, in previous versions this function does not exist — to make sure that the code works with all versions of QGIS, we first check whether the method exists.

Notitie: This method is deprecated as of QGIS 2.18.0 and will produce a warning. Simply calling `triggerRepaint()` is sufficient.

The second call emits signal that will force any map canvas containing the layer to issue a refresh.

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

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

In case you have changed layer symbology (see sections about raster and vector layers on how to do that), you might want to force QGIS to update the layer symbology in the layer list (legend) widget. This can be done as follows (`iface` is an instance of `QgisInterface`)

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

4.4 Waarden bevragen

To do a query on value of bands of raster layer at some specified point

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

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

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

CHAPTER 5

Vectorlagen gebruiken

- Informatie over attributen ophalen
- Objecten selecteren
- Itereren over vectorlagen
  - Toegang tot attributen
  - Itereren over geselecteerde objecten
  - Itereren over een deel van de objecten
- Vectorlagen bewerken
  - Objecten toevoegen
  - Objecten verwijderen
  - Objecten bewerken
  - Velden toevoegen en verwijderen
- Vectorlagen bewerken met een bewerkingsbuffer
- Ruimtelijke index gebruiken
- Writing Vector Layers
- Memory Provider
- Uiterlijk (symbologie) van vectorlagen
  - Renderer Enkel symbool
  - Renderer symbool Categoriën
  - Renderer symbool Gradueel
  - Werken met symbolen
    - Werken met symboollagen
    - Aangepaste typen voor symboollagen maken
  - Aangepaste renderers maken
- Meer onderwerpen

Dit gedeelte beschrijft verschillende acties die kunnen worden uitgevoerd met vectorlagen.

5.1 Informatie over attributen ophalen

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

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

Notitie: Starting from QGIS 2.12 there is also a `fields()` in `QgsVectorLayer` which is an alias to `pendingFields()`.

5.2 Objecten selecteren

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

To change the selection color you can use `setSelectionColor()` method of `QgsMapCanvas` as shown in the following example:

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

To add add features to the selected features list for a given layer, you can call `setSelectedFeatures()` passing to it the list of features IDs:

```
# 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()])
```

To clear the selection, just pass an empty list:

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

5.3 Itereren over vectorlagen

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

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

    # show some information about the feature
    if geom.type() == QGis.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == QGis.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == QGis.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 Toegang tot attributen

Attributes can be referred to by their name.

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

Alternatively, attributes can be referred to by index. This is will be a bit faster than using the name. For example, to get the first attribute:

```
print feature[0]
```

5.3.2 Itereren over geselecteerde objecten

if you only need selected features, you can use the `selectedFeatures()` method from vector layer:

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

Another option is the Processing `features()` method:

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

By default, this will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise. Note that this behavior can be changed in the Processing options to ignore selections.

5.3.3 Itereren over een deel van de objecten

If you want to iterate over a given subset of features in a layer, such as those within a given area, you have to add a `QgsFeatureRequest` object to the `getFeatures()` call. Here's an example

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

With `setLimit()` you can limit the number of requested features. Here's an example

```
request = QgsFeatureRequest()
request.setLimit(2)
for feature in layer.getFeatures(request):
    # loop through only 2 features
```

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the examples above, you can build an `QgsExpression` object and pass it to the `QgsFeatureRequest` constructor. Here's an example

```
# 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)
```

See *Expressies, filteren en waarden berekenen* for the details about the syntax supported by `QgsExpression`.

Het verzoek kan worden gebruikt om de gegevens per opgehaald object te definiëren, zodat de doorloop alle objecten retourneert, maar slechts een deel van de gegevens van elk daarvan teruggeeft.

```
# 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: Speed features request

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

5.4 Vectorlagen bewerken

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

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

For a list of all available capabilities, please refer to the API Documentation of QgsVectorDataProvider

To print layer's capabilities textual description in a comma separated list you can use `capabilitiesString()` as in the following example:

```
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'
```

Bij het gebruiken van de volgende methodes voor het bewerken van vectorlagen worden de wijzigingen direct opgeslagen in de onderliggende gegevensbron (een bestand, database etc.). Voor het geval u slechts tijdelijke wijzigingen wilt uitvoeren, ga dan naar het volgende gedeelte waarin uitgelegd wordt hoe *aanpassingen kunnen worden uitgevoerd met een bewerkingsbuffer*.

Notitie: Als u werkt binnen QGIS (ofwel vanuit de console of vanuit een plug-in), zou het nodig kunnen zijn het opnieuw tekenen van het kaartvenster te forceren om de wijzigingen te kunnen zien die u heeft gemaakt aan de geometrie, aan de stijl of aan de attributen:

```
# 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 Objecten toevoegen

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

To set up the attributes you can either initialize the feature passing a `QgsFields` instance or call `initAttributes()` passing the number of fields you want to be added.

```
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 Objecten verwijderen

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

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

5.4.3 Objecten bewerken

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

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

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

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

Tip: Favor QgsVectorLayerEditUtils class for geometry-only edits

If you only need to change geometries, you might consider using the `QgsVectorLayerEditUtils` which provides some of useful methods to edit geometries (translate, insert or move vertex etc.).

Tip: Directly save changes using `with` based command

Using `with edit(layer):` the changes will be commited automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollBack()` all the changes. See *Vectorlagen bewerken met een bewerkingsbuffer*.

5.4.4 Velden toevoegen en verwijderen

U moet een lijst met definities voor velden opgeven om velden toe te voegen (attributen). Geef een lijst met indexen van velden op om velden te verwijderen.

```
from PyQt4.QtCore import QVariant

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])
```

Na het verwijderen of toevoegen van velden in de gegevensprovider moeten de velden van de laag worden bijgewerkt omdat de wijzigingen niet automatisch worden doorgevoerd.

```
layer.updateFields()
```

5.5 Vectorlagen bewerken met een bewerkingsbuffer

When editing vectors within QGIS application, you have to first start editing mode for a particular layer, then do some modifications and finally commit (or rollback) the changes. All the changes you do are not written until you commit them — they stay in layer's in-memory editing buffer. It is possible to use this functionality also programmatically — it is just another method for vector layer editing that complements the direct usage of data providers. Use this option when providing some GUI tools for vector layer editing, since this will allow user to decide whether to commit/rollback and allows the usage of undo/redo. When committing changes, all changes from the editing buffer are saved to data provider.

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

```
from PyQt4.QtCore import QVariant

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

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

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

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

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

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

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

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

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

U kunt ook het argument `with edit(layer)`-gebruiken om commit en rollback in een meer semantisch codeblok op te nemen zoals weergegeven in het voorbeeld hieronder:

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

This will automatically call `commitChanges()` in the end. If any exception occurs, it will `rollBack()` all the changes. In case a problem is encountered within `commitChanges()` (when the method returns False) a `QgsEditError` exception will be raised.

5.6 Ruimtelijke index gebruiken

Ruimtelijke indexen kunnen de uitvoering van uw code enorm verbeteren als u frequent query's moet uitvoeren op een vectorlaag. Stel u bijvoorbeeld voor dat u een algoritme voor interpolatie schrijft, en dat voor een bepaalde locatie u de 10 dichtstbijzijnde punten van een puntenlaag wilt weten om die punten te gebruiken voor het berekenen van de waarde voor de interpolatie. Zonder een ruimtelijke index is de enige manier waarop QGIS die 10 punten kan vinden is door de afstand vanaf elk punt tot de gespecificeerde locatie te berekenen en dan die afstanden te vergelijken. Dit kan een zeer tijdrovende taak zijn, speciaal als het moet worden herhaald voor verschillende locaties. Als er een ruimtelijke index bestaat voor de laag, is de bewerking veel effectiever.

Denk aan een laag zonder ruimtelijke index als aan een telefoonboek waarin telefoonnummers niet zijn gesorteerd of geïndexeerd. De enige manier om het telefoonnummer van een bepaald persoon te vinden is door vanaf het begin te lezen totdat u het vindt.

Ruimtelijke indexen worden niet standaard gemaakt voor een vectorlaag in QGIS, maar u kunt ze eenvoudig maken. Dit is wat u dan moet doen:

- create spatial index — the following code creates an empty index

  ```
  index = QgsSpatialIndex()
  ```

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

  ```
  index.insertFeature(feat)
  ```

- als alternatief kunt u alle objecten van een laag in één keer laden met behulp van bulk laden

  ```
  index = QgsSpatialIndex(layer.getFeatures())
  ```

- als de ruimtelijke index eenmaal is gevuld met enkele waarden, kunt u enkele query's uitvoeren

  ```
  # 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 Writing Vector Layers

You can write vector layer files using `QgsVectorFileWriter` class. It supports any other kind of vector file that OGR supports (shapefiles, GeoJSON, KML and others).

There are two possibilities how to export a vector layer:

- from an instance of `QgsVectorLayer`

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

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

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

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

Voor geldige namen van stuurprogramma's, bekijk supported formats by OGR — u zou de waarde in de kolom "Code" door moeten geven als de naam van het stuurprogramma. Optioneel kunt u instellen om alleen geselecteerde objecten te exporteren, nadere specifieke opties voor het stuurprogramma voor het maken door te geven of de schrijver vertellen geen attributen te maken — bekijk de documentatie voor de volledige syntaxis.

- directly from features

```
from PyQt4.QtCore import QVariant

# 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 Sh

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 Memory Provider

Memory provider is intended to be used mainly by plugin or 3rd party app developers. It does not store data on disk, allowing developers to use it as a fast backend for some temporary layers.

De provider ondersteunt velden string, int en double.

The memory provider also supports spatial indexing, which is enabled by calling the provider's

`createSpatialIndex()` function. Once the spatial index is created you will be able to iterate over features within smaller regions faster (since it's not necessary to traverse all the features, only those in specified rectangle).

A memory provider is created by passing `"memory"` as the provider string to the `QgsVectorLayer` constructor.

De constructor accepteert ook een URI die het type geometrie van de laag definieert, één van: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, of `"MultiPolygon"`.

De URI mag ook het coördinaten referentiesysteem specificeren, velden, en indexeren van de memory-provider in de URI. De syntaxis is:

crs=definition Specifies the coordinate reference system, where definition may be any of the forms accepted by `QgsCoordinateReferenceSystem.createFromString()`

index=yes Specificeert dat de provider een ruimtelijke index zal gebruiken

field=name:type(length,precision) Specificeert een attribuut van de laag. Het attribuut heeft een naam en, optioneel, een type (integer, double of string), lengte en precisie. Er kunnen meerdere definities voor velden zijn.

Het volgende voorbeeld van een URI bevat al deze opties

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

De volgende voorbeeldcode illustreert het maken en vullen van een memory-provider

```
from PyQt4.QtCore import QVariant

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

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

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

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

Laten we tenslotte controleren of alles goed ging

```
# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMiniminum(), 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 Uiterlijk (symbologie) van vectorlagen

Wanneer een vectorlaag wordt gerenderd wordt het uiterlijk van de gegevens verschaft door de **renderer** en **symbolen** geassocieerd met de laag. Symbolen zijn klassen die zorg dragen voor het tekenen van visuele weergaven van objecten, terwijl renderers bepalen welk symbool zal worden gebruikt voor een bepaald object.

The renderer for a given layer can obtained as shown below:

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

En met die verwijzing, laten we het een beetje verkennen

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

There are several known renderer types available in QGIS core library:

Type	Klasse	Omschrijving
singleSymbol	QgsSingleSymbolRendererV2	Rendert alle objecten met hetzelfde symbool
categorizedSymbol	QgsCategorizedSymbolRendererV2	Rendert objecten door een ander symbool voor elke categorie te gebruiken
graduatedSymbol	QgsGraduatedSymbolRendererV2	Rendert objecten door een ander symbool voor elke bereik van waarden te gebruiken

There might be also some custom renderer types, so never make an assumption there are just these types. You can query `QgsRendererV2Registry` singleton to find out currently available renderers:

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

Het is mogelijk om een dump te verkrijgen van de inhoud van een renderer in de vorm van tekst — kan handig zijn bij debuggen

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

5.9.1 Renderer Enkel symbool

You can get the symbol used for rendering by calling `symbol()` method and change it with `setSymbol()` method (note for C++ devs: the renderer takes ownership of the symbol.)

You can change the symbol used by a particular vector layer by calling `setSymbol()` passing an instance of the appropriate symbol instance. Symbols for *point*, *line* and *polygon* layers can be created by calling the `createSimple()` function of the corresponding classes `QgsMarkerSymbolV2`, `QgsLineSymbolV2` and `QgsFillSymbolV2`.

The dictionary passed to `createSimple()` sets the style properties of the symbol.

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

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

`name` geeft de vorm van de markering aan, en kan één van de volgende zijn:

- `circle`

- square
- cross
- rectangle
- diamond
- pentagon
- triangle
- equilateral_triangle
- star
- regular_star
- arrow
- filled_arrowhead
- x

To get the full list of properties for the first symbol layer of a simbol instance you can follow the example code:

```
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'}
```

Dit kan nuttig zijn als u enkele eigenschappen wilt wijzigen:

```
# 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 Renderer symbool Categoriën

You can query and set attribute name which is used for classification: use `classAttribute()` and `setClassAttribute()` methods.

Een lijst categorieën verkrijgen

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

Where `value()` is the value used for discrimination between categories, `label()` is a text used for category description and `symbol()` method returns assigned symbol.

The renderer usually stores also original symbol and color ramp which were used for the classification: `sourceColorRamp()` and `sourceSymbol()` methods.

5.9.3 Renderer symbool Gradueel

Deze renderer lijkt erg veel op de renderer voor het symbool van de categorieën, hierboven beschreven, maar in plaats van één attribuutwaarde per klasse, werkt het met bereiken van waarden en kan dus alleen gebruikt worden met numerieke attributen.

Meer te weten komen over gebruikte bereiken in de renderer

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

U kunt opnieuw `classAttribute()` gebruiken om de naam van het attribuut voor classificatie te zoeken, methoden `sourceSymbol()` en `sourceColorRamp()`. Aanvullend is er de methode `mode()` die bepaalt hoe de bereiken werden gemaakt: met behulp van gelijke intervallen, kwantielen of een andere methode.

Als u uw eigen renderer voor symbolen Gradueel wilt maken, kunt u dat doen zoals is geïllustreerd in het voorbeeldsnippet hieronder (wat een eenvoudige schikking in twee klassen maakt)

```
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 Werken met symbolen

For representation of symbols, there is `QgsSymbolV2` base class with three derived classes:

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

Every symbol consists of one or more symbol layers (classes derived from `QgsSymbolLayerV2`). The symbol layers do the actual rendering, the symbol class itself serves only as a container for the symbol layers.

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

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

To find out symbol's color use `color()` method and `setColor()` to change its color. With marker symbols additionally you can query for the symbol size and rotation with `size()` and `angle()` methods, for line symbols there is `width()` method returning line width.

Grootte en breedte zijn standaard in millimeters, hoeken zijn in graden.

Werken met symboollagen

As said before, symbol layers (subclasses of `QgsSymbolLayerV2`) determine the appearance of the features. There are several basic symbol layer classes for general use. It is possible to implement new symbol layer types and thus arbitrarily customize how features will be rendered. The `layerType()` method uniquely identifies the symbol layer class — the basic and default ones are SimpleMarker, SimpleLine and SimpleFill symbol layers types.

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

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

Output

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

`QgsSymbolLayerV2Registry` class manages a database of all available symbol layer types.

To access symbol layer data, use its `properties()` method that returns a key-value dictionary of properties which determine the appearance. Each symbol layer type has a specific set of properties that it uses. Additionally, there are generic methods `color()`, `size()`, `angle()`, `width()` with their setter counterparts. Of course size and angle is available only for marker symbol layers and width for line symbol layers.

Aangepaste typen voor symboollagen maken

Veronderstel dat u de manier waarop gegevens worden gerenderd wilt aanpassen. U kunt uw eigen klasse voor de symboollaag maken dat de objecten op exact de wijze die u wilt tekent. Hier is een voorbeeld van een markering die rode cirkels met een gespecificeerde straal tekent

```
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)
```

The `layerType()` method determines the name of the symbol layer, it has to be unique among all symbol layers. Properties are used for persistence of attributes. `clone()` method must return a copy of the symbol layer with all attributes being exactly the same. Finally there are rendering methods: `startRender()` is called before rendering first feature, `stopRender()` when rendering is done. And `renderPoint()` method which does the rendering. The coordinates of the point(s) are already transformed to the output coordinates.

For polylines and polygons the only difference would be in the rendering method: you would use `renderPolyline()` which receives a list of lines, resp. `renderPolygon()` which receives list of points on outer ring as a first parameter and a list of inner rings (or None) as a second parameter.

Gewoonlijk is het handig om een GUI toe te voegen voor het instellen van attributen voor het type symboollaag om het voor gebruikers mogelijk te maken het uiterlijk aan te passen: in het geval van ons voorbeeld hierboven kunnen we de gebruiker de straal van de cirkel laten instellen. De volgende code implementeert een dergelijk 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()"))
```

Deze widget kan worden ingebed in het dialoogvenster van de eigenschappen voor het symbool. Wanneer het type symboollaag wordt geselecteerd in het dialoogvenster van de eigenschappen voor het symbool, maakt het een instance van de symboollaag en een instance van de widget van de symboollaag. Dan roept het de methode `setSymbolLayer()` aan om de symboollaag toe te wijzen aan de widget. In die methode zou de widget de UI moeten bijwerken om de attributen van de symboollaag weer te geven. De functie `symbolLayer()` wordt gebruikt om de symboollaag opnieuw op te halen bij het dialoogvenster Eigenschappen om het voor het symbool te gebruiken.

Bij elke wijziging van attributen zou de widget een signaal `changed()` moeten uitzenden om het dialoogvenster Eigenschappen de voorvertoning van het symbool bij te laten werken.

Nu missen we alleen nog de uiteindelijke lijm: om QGIS zich bewust te laten worden van deze nieuwe klassen. Dit wordt gedaan door de symboollaag toe te voegen aan het register. Het is mogelijk om de symboollaag ook te gebruiken zonder die toe te voegen aan het register, maar sommige functionaliteit zal niet werken: bijv. het laden van projectbestanden met de aangepaste symboollagen of de mogelijkheid om de attributen van de laag te bewerken in de GUI.

We zullen metadata moeten maken voor de symboollaag

```
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())
```

You should pass layer type (the same as returned by the layer) and symbol type (marker/line/fill) to the constructor of parent class. `createSymbolLayer()` takes care of creating an instance of symbol layer with attributes specified in the *props* dictionary. (Beware, the keys are QString instances, not “str” objects). And there is `createSymbolLayerWidget()` method which returns settings widget for this symbol layer type.

De laatste stap is om deze symboollaag toe te voegen aan het register — en we zijn klaar.

5.9.5 Aangepaste renderers maken

Het zou handig kunnen zijn om een nieuwe implementatie voor de renderer te maken als u de regels voor het selecteren van symbolen voor het renderen van objecten zou willen aanpassen. Sommige gebruiken gevallen waarin u dit zou willen doen: symbool wordt bepaald uit een combinatie van velden, grootte van symbolen wijzigt, afhankelijk van hun huidige schaal etc.

De volgende code geeft een eenvoudige aangepaste renderer weer die twee markeringssymbolen maakt en er, willekeurig, één kiest voor elk object

```
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.defaultSymbo

  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)
```

The constructor of parent `QgsFeatureRendererV2` class needs renderer name (has to be unique among renderers). `symbolForFeature()` method is the one that decides what symbol will be used for a particular feature. `startRender()` and `stopRender()` take care of initialization/finalization of symbol rendering. `usedAttributes()` method can return a list of field names that renderer expects to be present. Finally `clone()` function should return a copy of the renderer.

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

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2()
    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
```

The constructor receives instances of the active layer (`QgsVectorLayer`), the global style (`QgsStyleV2`) and current renderer. If there is no renderer or the renderer has different type, it will be replaced with our new renderer, otherwise we will use the current renderer (which has already the type we need). The widget contents should be updated to show current state of the renderer. When the renderer dialog is accepted, widget's `renderer()` method is called to get the current renderer — it will be assigned to the layer.

Het laatste ontbrekende gedeelte zijn de metadata voor de renderer en het registreren in het register, anders zal het

laden van de lagen met de renderer niet werken en zal de gebruiker niet in staat zijn die te selecteren uit de lijst met renderers. Laten we ons voorbeeld RandomRenderer voltooien

```
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())
```

Soortgelijk als met de symboollagen, verwacht de constructor voor abstracte metadata de naam van de renderer, de zichtbare naam voor de gebruikers en optioneel de naam van het pictogram voor de renderer. De methode `createRenderer()` geeft de instance `QDomElement` door die kan worden gebruikt om de status van de renderer opnieuw op te slaan in de boom van de DOM. De methode `createRendererWidget()` maakt het widget voor de configuratie. Die hoeft niet aanwezig te zijn of mag *None* teruggeven als de renderer geen GUI heeft.

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

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

The icon can be associated also at any later time using `setIcon()` method of the metadata class. The icon can be loaded from a file (as shown above) or can be loaded from a Qt resource (PyQt4 includes .qrc compiler for Python).

5.10 Meer onderwerpen

TODO:

- symbolen maken/aanpassen
- working with style (`QgsStyleV2`)
- working with color ramps (`QgsVectorColorRampV2`)
- rule-based renderer (see this blogpost)
- symboollaag en registraties van renderer verkennen

CHAPTER 6

Afhandeling van geometrie

- Construeren van geometrie
- Toegang tot geometrie
- Predicaten en bewerking voor geometrieën

Points, linestrings and polygons that represent a spatial feature are commonly referred to as geometries. In QGIS they are represented with the `QgsGeometry` class.

Soms is één geometrie in feite een verzameling van enkele (ééndelige) geometrieën. Een dergelijke geometrie wordt een geometrie met meerdere delen genoemd. Als het slechts één type eenvoudige geometrie bevat, noemen we het multi-punt, multi-lijn of multi-polygoon. Een land dat bijvoorbeeld bestaat uit meerdere eilanden kan worden weergegeven als een multi-polygoon.

De coördinaten van geometrieën kunnen in elk coördinaten referentiesysteem (CRS) staan. Bij het ophalen van objecten vanaf een laag, zullen de geassocieerde geometrieën in coördinaten in het CRS van de laag staan.

Description and specifications of all possible geometries construction and relationships are available in the OGC Simple Feature Access Standards for advanced details.

6.1 Construeren van geometrie

There are several options for creating a geometry:

- uit coördinaten

  ```
  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)]])
  ```

  Coordinates are given using `QgsPoint` class.

  Polyline (Linestring) is represented by a list of points. Polygon is represented by a list of linear rings (i.e. closed linestrings). First ring is outer ring (boundary), optional subsequent rings are holes in the polygon.

  Geometrieën die bestaan uit meerdere delen gaan een niveau verder: multi-punt is een lijst van punten, multi-lijnen zijn een lijst van lijnen en multi-polygoon is een lijst van polygonen.

- uit bekende tekst (WKT)

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

- uit bekende binaire (WKB)

  ```
  >>> g = QgsGeometry()
  >>> wkb = '010100000000000000000045400000000000001440'.decode('hex')
  >>> g.fromWkb(wkb)
  ```

```
>>> g.exportToWkt()
'Point (42 5)'
```

6.2 Toegang tot geometrie

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

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

As an alternative, one can use `type()` method which returns a value from `QGis.GeometryType` enumeration. There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

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

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

Notitie: The tuples (x,y) are not real tuples, they are `QgsPoint` objects, the values are accessible with `x()` and `y()` methods.

For multipart geometries there are similar accessor functions: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Predicaten en bewerking voor geometrieën

QGIS uses GEOS library for advanced geometry operations such as geometry predicates (`contains()`, `intersects()`, ...) and set operations (`union()`, `difference()`, ...). It can also compute geometric properties of geometries, such as area (in the case of polygons) or lengths (for polygons and lines)

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

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

Areas and perimeters don't take CRS into account when computed using these methods from the `QgsGeometry` class. For a more powerful area and distance calculation, the `QgsDistanceArea` class can be used. If projections are turned off, calculations will be planar, otherwise they'll be done on the ellipsoid.

```
d = QgsDistanceArea()
d.setEllipsoid('WGS84')
d.setEllipsoidalMode(True)
```

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

U kunt zoeken naar vele voorbeelden van algoritmen die zijn opgenomen in QGIS en die methoden gebruiken om vectorgegevens te analyseren en te transformeren. Hier zijn enkele koppelingen naar de code van sommige ervan.

Additional information can be found in following sources:

- Geometry transformation: Reproject algorithm
- Distance and area using the `QgsDistanceArea` class: Distance matrix algorithm
- Multi-part to single-part algorithm

CHAPTER 7

Ondersteuning van projecties

- Coördinaten ReferentieSystemen
- Projections

7.1 Coördinaten ReferentieSystemen

Coordinate reference systems (CRS) are encapsulated by `QgsCoordinateReferenceSystem` class. Instances of this class can be created by several different ways:

- specificeren van CRS met zijn ID

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

  QGIS gebruikt drie verschillende ID's voor elk referentiesysteem:

  - `PostgisCrsId` — code gebruikt in PostGIS databases.
  - `InternalCrsId` — interne QGIS code.
  - `EpsgCrsId` — code in de EPSG notatie

  Indien niet anders gespecificeerd in tweede parameter, wordt standaard PostGIS SRID gebruikt.

- specificeren van CRS door zijn well-known text (WKT)

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

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

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

It's wise to check whether creation (i.e. lookup in the database) of the CRS has been successful: `isValid()` must return `True`.

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

Accessing spatial reference system information

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.toProj4()
# 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 Projections

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

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

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

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

CHAPTER 8

Using Map Canvas

- Kaartvenster inbedden
- Gereedschappen voor de kaart gebruiken in het kaartvenster
- Elastieken banden en markeringen voor punten
- Aangepaste gereedschappen voor de kaart schrijven
- Aangepaste items voor het kaartvenster schrijven

The Map canvas widget is probably the most important widget within QGIS because it shows the map composed from overlaid map layers and allows interaction with the map and layers. The canvas shows always a part of the map defined by the current canvas extent. The interaction is done through the use of **map tools**: there are tools for panning, zooming, identifying layers, measuring, vector editing and others. Similar to other graphics programs, there is always one tool active and the user can switch between the available tools.

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

Whenever the map has been panned, zoomed in/out (or some other action triggers a refresh), the map is rendered again within the current extent. The layers are rendered to an image (using `QgsMapRenderer` class) and that image is then displayed in the canvas. The graphics item (in terms of the Qt graphics view framework) responsible for showing the map is `QgsMapCanvasMap` class. This class also controls refreshing of the rendered map. Besides this item which acts as a background, there may be more **map canvas items**. Typical map canvas items are rubber bands (used for measuring, vector editing etc.) or vertex markers. The canvas items are usually used to give some visual feedback for map tools, for example, when creating a new polygon, the map tool creates a rubber band canvas item that shows the current shape of the polygon. All map canvas items are subclasses of `QgsMapCanvasItem` which adds some more functionality to the basic `QGraphicsItem` objects.

Samenvattend, de architectuur van het kaartvenster bestaat uit drie concepten:

- kaartvenster — voor het bekijken van de kaart
- map canvas items — additional items that can be displayed in map canvas
- map tools — for interaction with map canvas

8.1 Kaartvenster inbedden

Kaartvenster is een widget net als elk ander widget van Qt, dus het gebruiken ervan is zo eenvoudig als het maken en weergeven ervan

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

This produces a standalone window with map canvas. It can be also embedded into an existing widget or window. When using .ui files and Qt Designer, place a `QWidget` on the form and promote it to a new class: set `QgsMapCanvas` as class name and set `qgis.gui` as header file. The `pyuic4` utility will take care of it. This is a very convenient way of embedding the canvas. The other possibility is to manually write the code to construct map canvas and other widgets (as children of a main window or dialog) and create a layout.

Standaard heeft kaartvenster een zwarte achtergrond en gebruikt geen anti-aliasing. Een witte achtergrond instellen en anti-aliasing inschakelen voor glad renderen

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

(In case you are wondering, `Qt` comes from `PyQt4.QtCore` module and `Qt.white` is one of the predefined `QColor` instances.)

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

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

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

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

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

Nadat deze opdrachten zijn uitgevoerd, zou het kaartvenster de laag moeten weergeven die u heeft geladen.

8.2 Gereedschappen voor de kaart gebruiken in het kaartvenster

The following example constructs a window that contains a map canvas and basic map tools for map panning and zooming. Actions are created for activation of each tool: panning is done with `QgsMapToolPan`, zooming in/out with a pair of `QgsMapToolZoom` instances. The actions are set as checkable and later assigned to the tools to allow automatic handling of checked/unchecked state of the actions – when a map tool gets activated, its action is marked as selected and the action of the previous map tool is deselected. The map tools are activated using `setMapTool()` method.

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

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

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

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

    self.setCentralWidget(self.canvas)

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

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

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

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

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

    self.pan()

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

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

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

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

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

Just make sure that the `mywnd.py` file is located within Python search path (`sys.path`). If it isn't, you can simply add it: `sys.path.insert(0, '/my/path')` — otherwise the import statement will fail, not finding the module.

8.3 Elastieken banden en markeringen voor punten

To show some additional data on top of the map in canvas, use map canvas items. It is possible to create custom canvas item classes (covered below), however there are two useful canvas item classes for convenience: `QgsRubberBand` for drawing polylines or polygons, and `QgsVertexMarker` for drawing points. They both work with map coordinates, so the shape is moved/scaled automatically when the canvas is being panned or zoomed.

Een polylijn weergeven

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

Een polygoon weregeven

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

Onthoud dat de punten voor polygoon geen platte lijst is: in feite is het een lijst van ringen die lineaire ringen van de polygoon bevat: de eerste ring is de buitenste grens, verdere (optionele) ringen corresponderen met gaten in de polygoon.

Elastieken banden maken enige aanpassingen mogelijk, namelijk om hun kleur en lijndikte te wijzigen

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

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

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

(in C++ is het mogelijk het item eenvoudigweg te verwijderen, in Python echter zou `del r` slechts de verwijzing verwijderen en zou het object nog steeds bestaan omdat het eigendom is van het kaartvenster)

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

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

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

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

For temporary hiding of vertex markers and removing them from canvas, the same applies as for the rubber bands.

8.4 Aangepaste gereedschappen voor de kaart schrijven

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

Map tools should inherit from the `QgsMapTool` class or any derived class, and selected as active tools in the canvas using the `setMapTool()` method as we have already seen.

Hier is een voorbeeld van een gereedschap voor de kaart dat het mogelijk maakt een rechthoekig bereik te definiëren door te klikken en te slepen in het kaartvenster. Wanneer de rechthoek is gedefinieerd, zal het de coördinaten voor de begrenzing afdrukken in de console. Het gebruikt de elementen voor elastieken banden zoals eerder beschreven om de geselecteerde rechthoek weer te geven als die wordt gedefinieerd.

```
class RectangleMapTool(QgsMapToolEmitPoint):
  def __init__(self, canvas):
      self.canvas = canvas
      QgsMapToolEmitPoint.__init__(self, self.canvas)
      self.rubberBand = QgsRubberBand(self.canvas, QGis.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 Aangepaste items voor het kaartvenster schrijven

TODO: hoe een item voor het kaartvenster te maken

```
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)
```

Kaart renderen en afdrukken

- Eenvoudig renderen
- Lagen met een verschillend CRS renderen
- Output using Map Composer
  - Output to a raster image
  - Output to PDF

There are generally two approaches when input data should be rendered as a map: either do it quick way using `QgsMapRenderer` or produce more fine-tuned output by composing the map with `QgsComposition` class and friends.

9.1 Eenvoudig renderen

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

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

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

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

render = QgsMapRenderer()

# set layer set
lst = [layer.id()]  # 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 Lagen met een verschillend CRS renderen

If you have more than one layer and they have a different CRS, the simple example above will probably not work: to get the right values from the extent calculations you have to explicitly set the destination CRS and enable OTF reprojection as in the example below (only the renderer configuration part is reported)

```
...
# 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 Output using Map Composer

Map composer is a very handy tool if you would like to do a more sophisticated output than the simple rendering shown above. Using the composer it is possible to create complex map layouts consisting of map views, labels, legend, tables and other elements that are usually present on paper maps. The layouts can be then exported to PDF, raster images or directly printed on a printer.

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

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

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

Note that the composition takes an instance of `QgsMapRenderer`. In the code we expect we are running within QGIS application and thus use the map renderer from map canvas. The composition uses various parameters from the map renderer, most importantly the default set of map layers and the current extent. When using composer in a standalone application, you can create your own map renderer instance the same way as shown in the section above and pass it to the composition.

It is possible to add various elements (map, label, ...) to the composition — these elements have to be descendants of `QgsComposerItem` class. Currently supported items are:

- kaart — dit item vertelt de bibliotheken waar de kaart zelf moet worden geplaatst. Hier maken we ene kaart en spreiden die over de gehele grootte van het papier

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

- label — maakt het weergeven van labels mogelijk. Het is mogelijk het lettertype, de kleur, de uitlijning en marge aan te passen

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

- legenda

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

- schaalbalk

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

- pijl
- afbeelding
- basisvorm
- op knopen gebaseerde vorm

```
polygon = QPolygonF()
polygon.append(QPointF(0.0, 0.0))
polygon.append(QPointF(100.0, 0.0))
polygon.append(QPointF(200.0, 100.0))
polygon.append(QPointF(100.0, 200.0))

composerPolygon = QgsComposerPolygon(polygon, c)
c.addItem(composerPolygon)

props = {}
props["color"] = "green"
props["style"] = "solid"
props["style_border"] = "solid"
props["color_border"] = "black"
props["width_border"] = "10.0"
props["joinstyle"] = "miter"

style = QgsFillSymbolV2.createSimple(props)
composerPolygon.setPolygonStyleSymbol(style)
```

- tabel

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

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

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

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

Besides creating the composer items by hand, QGIS has support for composer templates which are essentially compositions with all their items saved to a .qpt file (with XML syntax). Unfortunately this functionality is not yet available in the API.

Once the composition is ready (the composer items have been created and added to the composition), we can proceed to produce a raster and/or vector output.

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

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

9.3.1 Output to a raster image

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

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

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

# render the composition
imagePainter = QPainter(image)
c.renderPage( imagePainter, 0 )
imagePainter.end()

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

9.3.2 Output to PDF

The following code fragment renders a composition to a PDF file

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

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

CHAPTER 10

Expressies, filteren en waarden berekenen

- Parsen van expressies
- Evalueren van expressies
  - Basisexpressies
  - Expressies met objecten
  - Fouten afhandelen
- Voorbeelden

QGIS heeft enige ondersteuning voor het parsen van SQL-achtige expressies. Alleen een klein deel van de syntaxis voor SQL wordt ondersteund. De expressies kunnen worden geëvalueerd ófwel als Booleaanse uitdrukkingen (die True of False teruggeven) of als functies (die een scalaire waarde teruggeven). Bekijk *vector_expressions* in de Gebruikershandleiding voor een volledige lijst van beschikbare functies.

Drie basistypen worden ondersteund:

- number — zowel gehele getallen als decimale getallen, bijv. `123`, `3.14`
- string — zij moeten zijn omsloten door enkele aanhalingstekens: `'hallo wereld'`
- kolomverwijzing — tijdens evaluatie wordt de verwijzing vervangen door de actuele waarde van het veld. De namen worden niet geëscaped.

De volgende bewerkingen zijn beschikbaar:

- rekenkundige operatoren: `+`, `-`, `*`, `/`, `^`
- haakjes: voor het forceren van de voorrang van de operator: `(1 + 1) * 3`
- unaire plus en minus: `-12`, `+5`
- wiskundige functies: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- functies voor conversie: `to_int`, `to_real`, `to_string`, `to_date`
- geometrische functies: `$area`, `$length`
- functies voor afhandelen van geometrie: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

En de volgende termen worden ondersteund:

- vergelijking: =, != , >, >=, <, <=
- overeenkomst van patroon: `LIKE` (gebruiken van % en _), ~ (reguliere expressies)
- logische termen: `AND`, `OR`, `NOT`
- controle op waarde NULL: `IS NULL`, `IS NOT NULL`

Voorbeelden van termen:

- `1 + 2 = 3`
- `sin(hoek) > 0`

- `'Hallo' LIKE 'Ha%'`
- `(x > 10 AND y > 10) OR z = 0`

Voorbeelden van scalaire expressies:

- `2 ^ 10`
- `sqrt(waarde)`
- `$length + 1`

10.1 Parsen van expressies

```
>>> 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 Evalueren van expressies

10.2.1 Basisexpressies

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

10.2.2 Expressies met objecten

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

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

You can also use `QgsExpression.prepare()` if you need check more than one feature. Using `QgsExpression.prepare()` will increase the speed that evaluate takes to run.

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

10.2.3 Fouten afhandelen

```
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 Voorbeelden

Het volgende voorbeeld kan worden gebruikt om een laag te filteren en elk object terug te geven dat overeenkomt met een term.

```
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"
```

CHAPTER 11

Instellingen lezen en opslaan

Vaak is het voor een plug-in nuttig om enkele variabelen op te slaan zodat de gebruiker ze niet opnieuw hoeft in te voeren of te selecteren als de plug-in een volgende keer wordt uitgevoerd.

Deze variabelen kunnen worden opgeslagen en weer worden opgehaald met de hulp van Qt en de API van QGIS. Voor elke variabele zou u een sleutel moeten kiezen die kan worden gebruikt om toegang te verkrijgen tot de variabele — voor de favoriete kleur van de gebruiker zou u de sleutel "favourite_color" kunnen gebruiken of elke andere tekenreeks met betekenis. Het wordt aanbevolen enige structuur aan te brengen in het benoemen van sleutels.

We can make difference between several types of settings:

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

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

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

  The second parameter of the `value()` method is optional and specifies the default value if there is no previous value set for the passed setting name.

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

```
proj = QgsProject.instance()

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

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

As you can see, the `writeEntry()` method is used for all data types, but several methods exist for reading the setting value back, and the corresponding one has to be selected for each data type.

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

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

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

CHAPTER 12

Communiceren met de gebruiker

- Showing messages. The QgsMessageBar class
- Voortgang weergeven
- Loggen

Dit gedeelte geeft enkele methoden en elementen weer die zouden moeten worden gebruikt om te communiceren met de gebruiker, om consistentie in de gebruikersinterface te behouden.

12.1 Showing messages. The `QgsMessageBar` class

Het gebruiken van berichtenvakken kan een slecht idee zijn vanuit het gezichtspunt van de gebruiker. Voor het weergeven van een korte regel met informatie of een waarschuwing/foutberichten, is de QGIS berichtenbalk gewoonlijk een betere optie.

U kunt, met behulp van de verwijzing naar het interface-object van QGIS, een bericht weergeven in de berichtenbalk met de volgende code

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

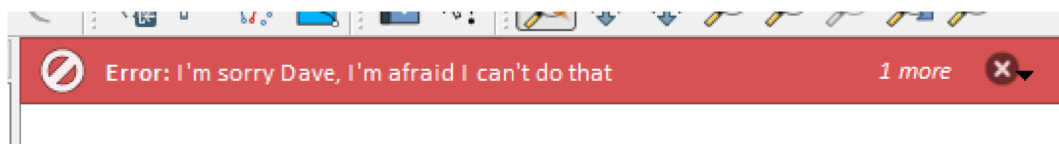


Figure 12.1: QGIS berichtenbalk

U kunt een duur instellen om het voor een beperkte tijd weer te geven

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

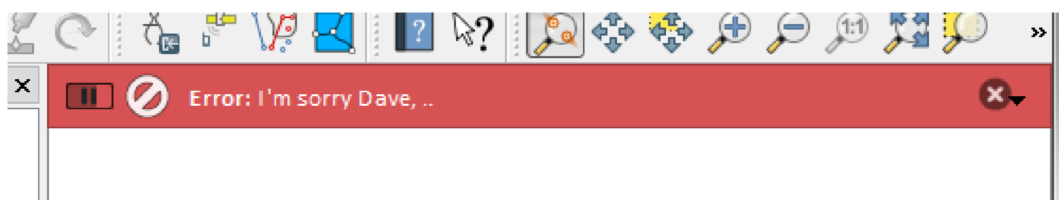


Figure 12.2: QGIS berichtenbalk met timer

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

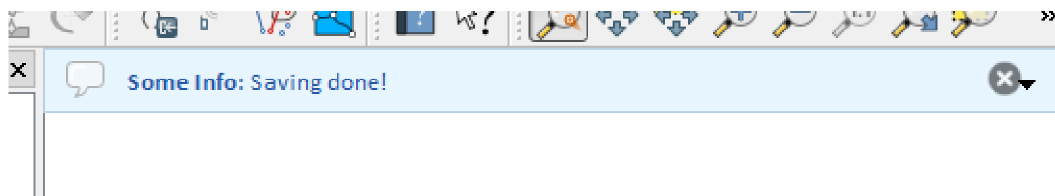


Figure 12.3: QGIS berichtenbalk (info)

Widgets kunnen aan de berichtenbalk worden toegevoegd, zoals bijvoorbeeld een knop om meer informatie weer te geven

```
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)
```

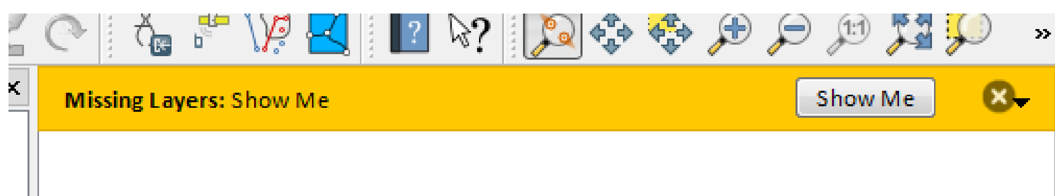


Figure 12.4: QGIS berichtenbalk met een knop

U kunt zelfs een berichtenbalk in uw eigen dialoogvenster gebruiken zodat u geen berichtenvak hoeft weer te geven, of als het geen zin heeft om het in het hoofdvenster van QGIS weer te geven

```
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 Voortgang weergeven

Voortgangsbalken kunnen ook worden opgenomen in de berichtenbalk van QGIS, omdat, zoals we al hebben gezien, die widgets accepteert. Hier is een voorbeeld dat u kunt proberen in de console.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
```

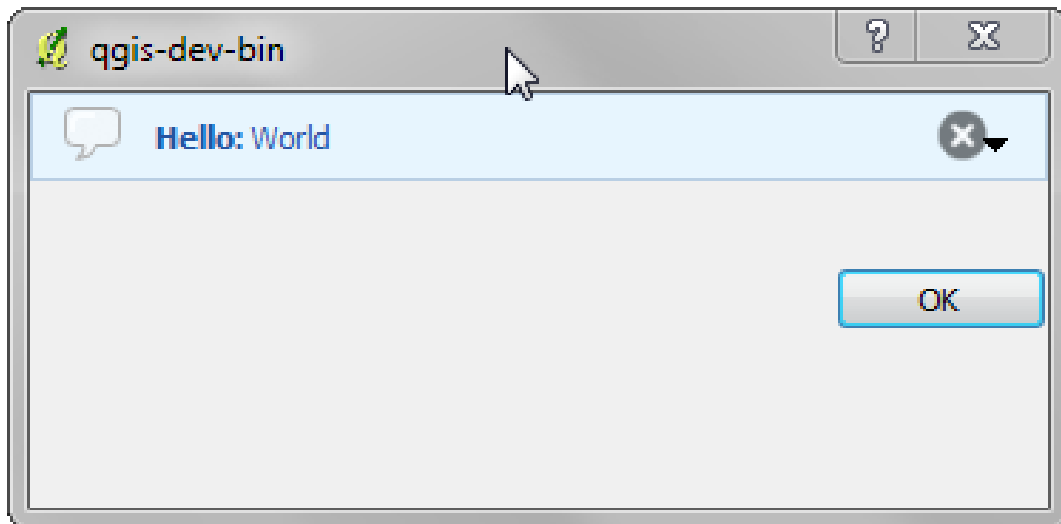


Figure 12.5: QGIS berichtenbalk in aangepast dialoogvenster

```
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

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

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

12.3 Loggen

U kunt het systeem voor loggen van QGIS gebruiken om alle informatie te loggen die u wilt opslaan over het uitvoeren van uw code.

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

Python plug-ins ontwikkelen

- Een plug-in schrijven
  - Plug-inbestanden
- Inhoud van de plug-in
  - Metadata van de plug-in
  - __init__.py
  - mainPlugin.py
  - Bronbestand
- Documentatie
- Vertaling
  - Software vereisten
  - Bestanden en map
    - .pro-bestand
    - .ts-bestand
    - .qm-bestand
  - Translate using Makefile
  - De plug-in laden

Het is mogelijk plug-ins te maken in de programmeertaal Python. In vergelijking met de klassieke plug-ins die zijn geschreven in C++ zouden deze eenvoudiger te schrijven, te begrijpen, te onderhouden en te verdelen zijn vanwege de dynamische natuur van de taal Python.

Plug-ins in Python worden samen met plug-ins in C++ vermeld in Beheer en installeer plug-ins in QGIS. Er wordt naar gezocht in deze paden:

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

De thuismap (hierboven vermeld als ~) op Windows is gewoonlijk iets als `C:\Documents and Settings\(gebruiker)` (op Windows XP of eerder) of `C:\Users\(gebruiker)`. Omdat QGIS Python 2.7 gebruikt, moeten submappen van deze paden een bestand __init__.py bevatten om te worden beschouwd als Python pakketten die kunnen worden geïmporteerd als plug-ins.

Notitie: Bij het instellen van *QGIS_PLUGINPATH* naar een bestaand pad voor een map, kunt u dit pad toevoegen aan de lijst met paden die wordt gebruikt voor het zoeken naar plug-ins.

Stappen:

1. *Idee*: Weet wat u met uw nieuwe plug-in voor QGIS wilt gaan doen. Waarom doet u dat? Welk probleem wilt u oplossen? Is er al een andere plug-in voor dat probleem?
2. *Bestanden maken*: Maak de hieronder beschreven bestanden. Een beginpunt (`__init__.py`). Vul *Metadata van de plug-in* (`metadata.txt`) in. Een hoofdgedeelte voor een plug-in in Python (`mainplugin.py`). Een formulier in QT-Designer (`form.ui`), met zijn `resources.qrc`.

3. *Code schrijven*: Schrijf de code in `mainplugin.py`
4. *Testen*: Sluit en heropen QGIS en importeer uw plug-in opnieuw. Controleer of alles OK is.
5. *Publiceren*: Publiceer uw plug-in in de opslagplaats van QGIS of maak uw eigen opslagplaats als een "arsenaal" van persoonlijke "wapens voor GIS".

13.1 Een plug-in schrijven

Sinds de introductie van plug-ins in Python in QGIS, zijn een anatal Plug-ins verschenen - op Plugin Repositories wiki page kunt u er enkele van vinden, u kunt hun bronnen gebruiken om meer te leren over het programmeren met PyQGIS of uitzoeken of u de inspanningen voor de ontwikkeling niet dupliceert. Het team van QGIS onderhoudt ook een *Officiële Python plug-in opslagplaats*. Klaar om een plug-in te maken, maar geen idee wat te doen? Python Plugin Ideas wiki page vermeldt wensen van de gemeenschap!

13.1.1 Plug-inbestanden

Hier is de mappenstructuur van uw voorbeeld-plug-in

```
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*
```

Wat is de betekenis van de bestanden:

- `__init__.py` = Het beginpunt van de plug-in. Het moet de methode `classFactory()` hebben en mag elke andere code voor initialisatie hebben.
- `mainPlugin.py` = De belangrijkste werkende code van de plug-in. Bevat alle informatie over de acties van de plug-in en de hoofdcode.
- `resources.qrc` = Het door Qt Designer gemaakte .xml-document. Bevat relatieve paden naar de bronnen van de formulieren.
- `resources.py` = De vertaling van het bestand .qrc, hierboven beschreven, naar Python.
- `form.ui` = De GUI, gemaakt door Qt Designer.
- `form.py` = De vertaling van de form.ui, hierboven beschreven, naar Python.
- `metadata.txt` = Vereist voor QGIS >= 1.8.0. bevat algemene informatie, versie, naam en enkele andere metadata, gebruikt door de website van de plug-in en infrastructuur van de plug-in. Vanaf QGIS 2.0 wordt de metadata uit `__init__.py` niet meer geaccepteerd en is het bestand `metadata.txt` vereist.

Hier staat een online geautomatiseerde manier voor het maken van de basisbestanden (skelet) van een typische plug-in voor Python in QGIS.

Er is ook een plug-in voor QGIS, genaamd Plugin Builder die een sjabloon voor een plug-in maakt uit QGIS en geen internetverbinding vereist. Dit is de aanbevolen optie, omdat het compatibele bronnen produceert voor 2.0.

Waarschuwing: Als u van plan bent de plug-in te uploaden naar *Officiële Python plug-in opslagplaats* moet u controleren of uw plug-in enkele aanvullende regels volgt, vereist voor plug-in *Validatie*

13.2 Inhoud van de plug-in

Hier vindt u informatie en voorbeelden over wat in elk van de bestanden moet worden toegevoegd in de hierboven beschreven bestandsstructuur.

13.2.1 Metadata van de plug-in

Als eerste moet beheer en installeer plug-ins enige basisinformatie ophalen over de plug-in, zoals de naam, omschrijving etc. ervan. Bestand `metadata.txt` is de juiste plaats om deze informatie te vermelden.

Belangrijk: Alle metadata moet inde codering UTF-8 zijn.

Naam van de metadata	Vereist	Opmerkingen
name	Ja	een korte string die de naam van de plug-in bevat
qgisMinimumVersion	Ja	gestippelde notatie van de minimale versie van QGIS
qgisMaximumVersion	Nee	gestippelde notatie van de maximale versie van QGIS
description	Ja	korte tekst die de plug-in beschrijft, geen HTML toegestaan
about	Ja	langere tekst die de plug-in tot in detail beschrijft, geen HTML toegestaan
version	Ja	korte string met de versie in gestippelde notatie
author	Ja	author name
email	Ja	e-mail van de auteur, niet weergegeven in Plug-ins beheren en installeren van QGIS of op de website, tenzij door een geregistreerde ingelogde gebruiker, dus alleen zichtbaar voor andere auteurs van plug-ins en beheerders van de website voor plug-ins
changelog	Nee	string, mag meerdere regels zijn, geen HTML toegestaan
experimental	Nee	Booleaanse vlag, *True* of *False*
deprecated	Nee	Booleaanse vlag, *True* of *False*, is van toepassing op de gehele plug-in en niet alleen op de geüploade versie
tags	Nee	kommagescheiden lijst, spaties zijn binnen de individuele tags toegestaan
homepage	Nee	een geldige URL die verwijst naar de startpagina voor uw plug-in
repository	Ja	een geldige URL voor de opslagplaats van de broncode
tracker	Nee	een geldige URL voor tickets en probleemrapporten
icon	Nee	een bestandsnaam of een relatief pad (relatief ten opzichte van de basismap van het gecomprimeerde pakket van de plug-in) of een webvriendelijke afbeelding (PNG, JPEG)
category	Nee	één van *Raster*, *Vector*, *Database* of *Web*

Standaard worden plug-ins geplaatst in het menu *Plug-ins* (we zullen in het volgende gedeelte zien hoe een menu-item voor uw plug-in toe te voegen) maar zij kunnen ook worden geplaatst in de menu's *Raster*, *Vector*, *Database* en *Web*.

Een overeenkomend item voor de metadata "category" bestaat om dat te specificeren, zodat de plug-in overeenkomstig kan worden geclassificeerd. Dit item voor de metadata wordt gebruikt als tip voor de gebruikers en vertelt ze waar (in welk menu) de plug-in kan worden gevonden. Toegestane waarden voor "category" zijn: Vector, Raster, Database of Web. Als u bijvoorbeeld wilt dat uw plug-in bereikbaar is in het menu *Raster*, voeg dat dan toe aan `metadata.txt`

```
category=Raster
```

Notitie: Als *qgisMaximumVersion* leeg is, zal het automatisch worden ingesteld op de hoofdversie plus *.99* indien geüpload naar de *Officiële Python plug-in opslagplaats*.

Een voorbeeld voor dit 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

Dit bestand wordt vereist door het systeem voor importeren van Python. Ook vereist QGIS dat dit bestand een functie `classFactory()` bevat, die wordt aangeroepen als de plug-in wordt geladen in QGIS. Het ontvangt een verwijzing naar de instance van `QgisInterface` en moet een instance teruggeven van de klasse van uw plug-in uit `mainplugin.py` — in ons geval is dat genaamd `TestPlugin` (zie hieronder). Zo zou `__init__.py` er uit moeten zien

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

## any other initialisation needed
```

13.2.3 mainPlugin.py

Dit is waar de magie gebeurt en dit is hoe de magie eruit ziet: (bijjv. `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.mainWinc
    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 form signal of the canvas
    QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTe

  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!"
```

De enige functies voor plug-ins die moeten bestaan in het hoofd-bronbestand (bijv. `mainPlugin.py`) zijn:

- `__init__` –> wat toegang geeft tot de interface van QGIS
- `initGui()` –> aangeroepen wanneer de plug-in wordt geladen
- `unload()` –> aangeroepen wanneer de plug-in wordt ontladen

U kunt zien dat in het voorbeeld hierboven `addPluginToMenu()` is gebruikt. Dit zal de overeenkomstige menuactie toevoegen aan het menu *Plug-ins*. Alternatieve methoden bestaan om de actie aan een ander menu toe te wijzen. Hier is een lijst met die methoden:

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

- addPluginToWebMenu()

Alle hebben dezelfde syntaxis als de methode addPluginToMenu().

Toevoegen van het menu van uw plug-in aan een van de voorgedefinieerde methoden wordt aanbevolen om consistentie te behouden in hoe items voor plug-ins zijn georganiseerd. U kunt echter uw aangepaste groepen voor het menu direct aan de Menubalk toevoegen, zoals het volgende voorbeeld demonstreert:

```
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.mainWin
    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()
```

Vergeet niet om QAction en QMenu objectName in te stellen op een naam die specifiek is voor uw plug-in zodat hij kan worden aangepast.

13.2.4 Bronbestand

U kunt zien dat we in initGui() een pictogram hebben gebruikt uit het bronbestand (in ons geval resources.qrc aangeroepen)

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

Het is goed om een voorvoegsel te gebruiken dat niet zal botsen met andere plug-ins of andere delen van QGIS, anders krijgt u misschien bronnen die u niet wilt. Nu dient nog slechts een bestand in Python te genereren dat de bronnen zal bevatten. dat wordt gedaan met de opdracht **pyrcc4**

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

Notitie: In omgevingen van Windows zal het proberen uit te voeren van de opdracht **pyrcc4** vanuit de Opdracht prompt of Powershell resulteren in de fout “Windows cannot access the specified device, path, or file [...]”. De eenvoudigste oplossing is waarschijnlijk om de OSGeo4W Shell te gebruiken maar als u er niet voor terugschrikt om de omgevingsvariabele PATH aan te passen of het pad naar het uitvoerbare bestand expliciet te specificeren zou u in staat moeten zijn het te vinden op <Your QGIS Install Directory>\bin\pyrcc4.exe.

En dat is alles... niets gecompliceerds :)

Als u alles juist heeft gedaan zou u in staat moeten zijn uw plug-in op te zoeken en te laden vanuit Beheer en installeer plug-ins en een bericht in de console te zien wanneer het pictogram op de werkbalk of het menuitem is geselecteerd.

Bij het werken aan een echte plug-in is het verstandig om de plug-in in een andere (werk-)map te schrijven en een makefile te maken dat de UI + bronbestanden zal genereren en de plug-in zal installeren in uw installatie van QGIS.

13.3 Documentatie

De documentatie voor de plug-in mag worden geschreven als helpbestanden in HTML. De module `qgis.utils` verschaft een functie, `showPluginHelp()` dat de browser voor Helpbestanden zal openen, op dezelfde manier als andere help voor QGIS.

De functie `showPluginHelp`()` zoekt naar de helpbestanden in dezelfde map als waar de module wordt aangeroepen. Het zal zoeken naar, op volgorde, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` en `index.html`, en geeft die, welke als eerste wordt gevonden, weer. Hier is `ll_cc` de locale van QGIS. Dit maakt het mogelijk meerdere vertalingen van de documentatie op te nemen met de plug-in.

De functie `showPluginHelp()` kan ook de parameters packageName , welke een specifieke plug-in specificeert waarvoor de Helpbestanden zullen worden weergegeven, filename, wat "index" mag vervangen in de namen van de gezochte bestanden, en section, wat de naam is van een tag voor een HTML-anker in het document waar de browser zal worden gepositioneerd, aannemen.

13.4 Vertaling

Met enkele stappen kunt u de omgeving instellen voor de vertaling van de plug-in zodat, afhankelijk van de instellingen voor de locale op uw computer, zal de plug-in worden geladen in verschillende talen.

13.4.1 Software vereisten

De eenvoudigste manier om alle bestanden voor vertalingen te maken en te beheren is om Qt Linguist te installeren. In een op Debian gebaseerde GNU/ Linux-achtige omgeving kunt u het installeren door te typen:

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

13.4.2 Bestanden en map

Wanneer u de plug-in maakt vindt u de map `i18n` in de hoofdmap van de map.

Alle bestanden voor de vertalingen moeten in deze map staan.

.pro-bestand

Eerst zou u een `.pro`-bestand moeten maken, dat is een *project*-bestand dat kan worden beheerd door **Qt Linguist**.

In dit `.pro`-bestand dient u alle bestanden en formulieren te specificeren die u wilt vertalen. Dit bestand wordt gebruikt om de bestanden en variabelen voor de vertalingen in te stellen. Een mogelijk projectbestand dat overeenkomt met de structuur van onze *example plugin*:

```
FORMS = ../form.ui
SOURCES = ../your_plugin.py
TRANSLATIONS = your_plugin_it.ts
```

Uw plug-in zou een meer complexe structuur kunnen hebben, en het zou uit meerdere verschillende bestanden kunnen bestaan. Als dat het geval is, onthoud dan dat `pylupdate4`, het programma dat we gebruiken om het `.pro`-bestand te lezen en de te vertalen tekenreeksen bij te werken, geen jokertekens toestaat, dus dient u elk bestand expliciet te vermelden in het `.pro`-bestand. Uw projectbestand zou er dan mogelijk als volgt uit kunnen zien:

```
FORMS = ../ui/about.ui ../ui/feedback.ui \
        ../ui/main_dialog.ui
SOURCES = ../your_plugin.py ../computation.py \
          ../utils.py
```

Verder is het bestand `your_plugin.py` het bestand dat alle menu's en sub-menu's van uw plug-in in de werkbalk van QGIS *aanroept* en u wilt het in zijn geheel vertalen.

Tenslotte kunt u met de variabele *TRANSLATIONS* de vertaalde talen specificeren die u wilt.

> **Waarschuwing:** Zorg er voor het bestand `ts` net zo te noemen als `your_plugin_` + `language` + `.ts` anders zal het laden van de taal mislukken! Gebruik 2-letterige afkortingen voor de taal (**it** voor Italiaans, **de** voor Duits, etc...)

.ts-bestand

Als u eenmaal de `.pro` hebt gemaakt bent u gereed om de/het `.ts` bestand(en) van de taal(talen) voor uw plug-in te genereren.

Open een terminal, ga naar de map `your_plugin/i18n` en type:

```
pylupdate4 your_plugin.pro
```

u zou het/de bestand(en) `your_plugin_language.ts` moeten zien.

Open het bestand `.ts` met **Qt Linguist** en begin met vertalen.

.qm-bestand

Wanneer het vertalen van uw plug-in voltooid is (als enkele tekenreeksen niet voltooid zijn zal de taal van de bron worden gebruikt voor die tekenreeksen) dient u het bestand `.qm` te maken (het gecompileerde `.ts`-bestand dat zal worden gebruikt door QGIS).

Open een terminal, cd naar de map `your_plugin/i18n` en type:

```
lrelease your_plugin.ts
```

nu zou u in de map `i18n` het/de bestand(en) `your_plugin_language.ts` moeten zien.

13.4.3 Translate using Makefile

Alternatively you can use the makefile to extract messages from python code and Qt dialogs, if you created your plugin with Plugin Builder. At the beginning of the Makefile there is a LOCALES variable:

```
LOCALES = en
```

Add the abbreviation of the language to this variable, for example for Hungarian language:

```
LOCALES = en hu
```

Now you can generate or update the `hu.ts` file (and the `en.ts` too) from the sources by:

```
make transup
```

After this, you have updated `.ts` file for all languages set in the LOCALES variable. Use **Qt4 Linguist** to translate the program messages. Finishing the translation the `.qm` files can be created by the transcompile:

```
make transcompile
```

You have to distribute `.ts` files with your plugin.

13.4.4 De plug-in laden

Open QGIS, wijzig de taal (*Extra* → *Opties* → *Taal*) en herstart QGIS om de vertaling van uw plug-in te zien.

U zou uw plug-in in de juiste taal moeten zien.

> **Waarschuwing:** Indien u iets wijzigt in uw plug-in (nieuwe UI's, nieuw menu, etc..) dient de bijgewerkte versie van de bestanden `.ts` en `.qm` **opnieuw te generen**, voer dus opnieuw de hierboven genoemde opdracht uit.

CHAPTER 14

Infrastructuur voor authenticatie

- Introductie
- Woordenlijst
- QgsAuthManager is het toegangspunt
  - Initialiseren van de beheerder en het hoofdwachtwoord instellen
  - Vullen van authdb met een nieuw item Configuratie voor authenticatie
    - Beschikbare authenticatiemethoden
    - Autoriteiten vullen
    - PKI-bundels beheren met QgsPkiBundle
  - Item verwijderen uit authdb
  - Uitbreiden van authcfg overlaten aan QgsAuthManager
    - PKI-voorbeelden met andere gegevensproviders
- Plug-ins aanpassen om de infrastructuur voor authenticatie te gebruiken
- GUI's voor authenticatie
  - GUI om persoonlijke gegevens te selecteren
  - Bewerkers voor GUI authenticatie
  - Bewerker voor GUI autoriteiten

14.1 Introductie

Verwijzingen voor de gebruiker voor de infrastructuur voor authenticatie kan worden nagelezen in de Gebruikershandleiding in het gedeelte *authentication_overview*.

Dit hoofdstuk beschrijft de beste praktijken om het systeem voor authenticatie te gebruiken uit het perspectief van de ontwikkelaar.

> **Waarschuwing:** Authentication system API is more than the classes and methods exposed here, but it's strongly suggested to use the ones described here and exposed in the following snippets for two main reasons
>
> 1. Authentication API will change during the move to QGIS3
> 2. Python bindings will be restricted to the `QgsAuthManager` class use.

De meeste van de volgende snippets zijn afgeleid van de code voor de plug-in Geoserver Explorer en de testen daarvan. Dit is de eerste plug-in die de infrastructuur voor authenticatie gebruikte. De code voor de plug-in en de testen daarvan is te vinden via deze link. Andere goede verwijzingen naar code zijn te lezen in de infrastructuur voor authenticatie tests code

14.2 Woordenlijst

Hier zijn enkele definities van de meest voorkomende objecten die worden behandeld in dit hoofdstuk.

Hoofdwachtwoord Wachtwoord voor toegang tot en ontsleutelen van gegevens die zijn opgeslagen in de QGIS Authentication DB

Authenticatie-database A *Master Password* crypted sqlite db `<user home>/.qgis2/qgis-auth.db` where *Authentication Configuration* are stored. e.g user/password, personal certificates and keys, Certificate Authorities

Authenticatie DB *Authentication Database*

Configuratie voor authenticatie Een set van gegevens voor authenticatie afhankelijk van de *Authentication Method*. bijv de methode Basisauthenticatie slaat het paar gebruiker/wachtwoord op.

Configuratie voor authenticatie *Authentication Configuration*

Authenticatiemethode Een specifieke methode die wordt gebruikt om te worden geauthenticeerd. Elke methode heeft zijn eigen protocol dat wordt gebruikt om het bepaalde niveau voor authenticatie te verkrijgen. Elke methode is geïmplementeerd als gedeelde bibliotheek die dynamisch wordt geladen gedurende de init van de infrastructuur voor authenticatie van QGIS.

14.3 QgsAuthManager is het toegangspunt

The QgsAuthManager singleton is the entry point to use the credentials stored in the QGIS encrypted *Authentication DB*:

```
<user home>/.qgis2/qgis-auth.db
```

Deze klasse zorgt voor de interactie met de gebruiker: door te vragen het hoofdwachtwoord in te stellen of door het transparant te gebruiken voor toegang tot opgeslagen versleutelde informatie.

14.3.1 Initialiseren van de beheerder en het hoofdwachtwoord instellen

Het volgende snippet geeft een voorbeeld om het hoofdwachtwoord in te stellen om toegang te krijgen tot de instellingen voor authenticatie. Opmerkingen in de code zijn belangrijk om het snippet te begrijpen.

```
authMgr = QgsAuthManager.instance()
# check if QgsAuthManager has been already initialized... a side effect
# of the QgsAuthManager.init() is that AuthDbPath is set.
# QgsAuthManager.init() is executed during QGis application init and hence
# you do not normally need to call it directly.
if authMgr.authenticationDbPath():
    # already initilised => we are inside a QGIS app.
    if authMgr.masterPasswordIsSet():
        msg = 'Authentication master password not recognized'
        assert authMgr.masterPasswordSame( "your master password" ), msg
    else:
        msg = 'Master password could not be set'
        # The verify parameter check if the hash of the password was
        # already saved in the authentication db
        assert authMgr.setMasterPassword( "your master password",
                                          verify=True), msg
else:
    # outside qgis, e.g. in a testing environment => setup env var before
    # db init
    os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
    msg = 'Master password could not be set'
    assert authMgr.setMasterPassword("your master password", True), msg
    authMgr.init( "/path/where/located/qgis-auth.db" )
```

14.3.2 Vullen van authdb met een nieuw item Configuratie voor authenticatie

Any stored credential is a *Authentication Configuration* instance of the QgsAuthMethodConfig class accessed using a unique string like the following one:

```
authcfg = 'fm1s770'
```

die string wordt automatisch gegenereerd bij het maken van een item met behulp van de API van QGIS of de GUI.

QgsAuthMethodConfig is the base class for any *Authentication Method*. Any Authentication Method sets a configuration hash map where authentication informations will be stored. Hereafter an useful snippet to store PKI-path credentials for an hypothetic alice user:

```
authMgr = QgsAuthManager.instance()
# set alice PKI data
p_config = QgsAuthMethodConfig()
p_config.setName("alice")
p_config.setMethod("PKI-Paths")
p_config.setUri("http://example.com")
p_config.setConfig("certpath", "path/to/alice-cert.pem" ))
p_config.setConfig("keypath", "path/to/alice-key.pem" ))
# check if method parameters are correctly set
assert p_config.isValid()

# register alice data in authdb returning the ''authcfg'' of the stored
# configuration
authMgr.storeAuthenticationConfig(p_config)
newAuthCfgId = p_config.id()
assert (newAuthCfgId)
```

Beschikbare authenticatiemethoden

*Authentication Method*s worden dynamisch geladen gedurende de initialisatie van de beheerder voor de authenticatie. De lijst van authenticatiemethoden kan variëren met de evolutie van QGIS, maar de originele lijst met beschikbare methoden is:

1. `Basic` Gebruiker en wachtwoordauthenticatie
2. `Identity-Cert` Authenticatie met Identiteitscertificaat
3. `PKI-Paths` Authenticatie met PKI-paden
4. `PKI-PKCS#12` Authenticatie met PKI PKCS#12

De bovenstaande tekenreeksen identificeren de authenticatiemethoden het het systeem voor authenticatie van QGIS. In het gedeelte Development wordt beschreven hoe een nieuwe C++ *Authentication Method*te maken.

Autoriteiten vullen

```
authMgr = QgsAuthManager.instance()
# add authorities
cacerts = QSslCertificate.fromPath( "/path/to/ca_chains.pem" )
assert cacerts is not None
# store CA
authMgr.storeCertAuthorities(cacerts)
# and rebuild CA caches
authMgr.rebuildCaCertsCache()
authMgr.rebuildTrustedCaCertsCache()
```

Waarschuwing: Vanwege beperkingen in de interface QT4/OpenSSL, worden bijgewerkte gecachte CA's ongeveer een minuut later weergegeven in OpenSsl. Hopelijk zal dit zijn opgelost in de infrastructuur voor authenticatie in QT5.

PKI-bundels beheren met QgsPkiBundle

A convenience class to pack PKI bundles composed on SslCert, SslKey and CA chain is the QgsPkiBundle class. Hereafter a snippet to get password protected:

```
# add alice cert in case of key with pwd
boundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                     "/path/to/alice-key_w-pass.pem",
                                     "unlock_pwd",
                                     "list_of_CAs_to_bundle" )
assert boundle is not None
assert boundle.isValid()
```

Refer to QgsPkiBundle class documentation to extract cert/key/CAs from the bundle.

14.3.3 Item verwijderen uit authdb

We kunnen een item verwijderen uit de *Authentication Database* met behulp van zijn identificatie `authcfg` met het volgende snippet:

```
authMgr = QgsAuthManager.instance()
authMgr.removeAuthenticationConfig( "authCfg_Id_to_remove" )
```

14.3.4 Uitbreiden van authcfg overlaten aan QgsAuthManager

De beste manier om een in de *Authentication DB* opgeslagen *Authentication Config* te gebruiken is door er naar te verwijzen met de unieke identificatie `authcfg`. Uitbreiden ervan betekent het converteren van een identificatie naar een volledige set van gegevens. De beste praktijk om opgeslagen *Authentication Config*s te gebruiken, is om het automatisch te laten worden beheerd door de beheerder van de Authenticatie. Het meest voorkomende gebruik van een opgeslagen configuratie is om te verbinden met een met authenticatie ingeschakelde service zoals een WMS of WFS of naar een verbinding met een DB.

Notitie: Take into account that not all QGIS data providers are integrated with the Authentication infrastructure. Each authentication method, derived from the base class QgsAuthMethod and support a different set of Providers. For example `Identity-Cert` method supports the following list of providers:

```
In [19]: authM = QgsAuthManager.instance()
In [20]: authM.authMethod("Identity-Cert").supportedDataProviders()
Out[20]: [u'ows', u'wfs', u'wcs', u'wms', u'postgres']
```

Voor toegang tot, bijvoorbeeld, een WMS-service met behulp van de opgeslagen gegevens die worden geïdentificeerd als `authcfg = 'fm1s770'`, hoeven we slechts de `authcfg` te gebruiken in de URL voor de gegevensbron zoals in het volgende snippet:

```
authCfg = 'fm1s770'
quri = QgsDataSourceURI()
quri.setParam("layers", 'usa:states')
quri.setParam("styles", '')
quri.setParam("format", 'image/png')
quri.setParam("crs", 'EPSG:4326')
quri.setParam("dpiMode", '7')
quri.setParam("featureCount", '10')
quri.setParam("authcfg", authCfg)   # <---- here my authCfg url parameter
quri.setParam("contextualWMSLegend", '0')
quri.setParam("url", 'https://my_auth_enabled_server_ip/wms')
rlayer = QgsRasterLayer(quri.encodedUri(), 'states', 'wms')
```

In het bovenstaande geval zal de provider `wms` er zorg voor dragen dat parameter voor de URI `authcfg` wordt uitgebreid met persoonlijke gegevens net vóór het instellen van de verbinding met HTTP.

Waarschuwing: Developer would have to leave `authcfg` expansion to the QgsAuthManager, in this way he will be sure that expansion is not done too early.

Usually an URI string, build using `QgsDataSourceURI` class, is used to set QGIS data source in the following way:

```
rlayer = QgsRasterLayer( quri.uri(False), 'states', 'wms')
```

Notitie: De parameter `False` is belangrijk om volledige uitbreiding van de ID voor `authcfg`, die aanwezig is in de URI, te voorkomen.

PKI-voorbeelden met andere gegevensproviders

Andere voorbeelden kunnen direct worden ingelezen in de QGIS tests upstream zoals in test_authmanager_pki_ows of test_authmanager_pki_postgres.

14.4 Plug-ins aanpassen om de infrastructuur voor authenticatie te gebruiken

Many third party plugins are using httplib2 to create HTTP connections instead of integrating with `QgsNetworkAccessManager` and its related Authentication Infrastructure integration. To facilitate this integration an helper python function has been created called `NetworkAccessManager`. Its code can be found here.

Deze hulpklasse kan worden gebruikt zoals in het volgende snippet:

```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 GUI's voor authenticatie

In deze alinea zijn de beschikbare GUI's vermeld die handig zijn om de infrastructuur voor authenticatie te integreren in aangepaste/eigen interfaces.

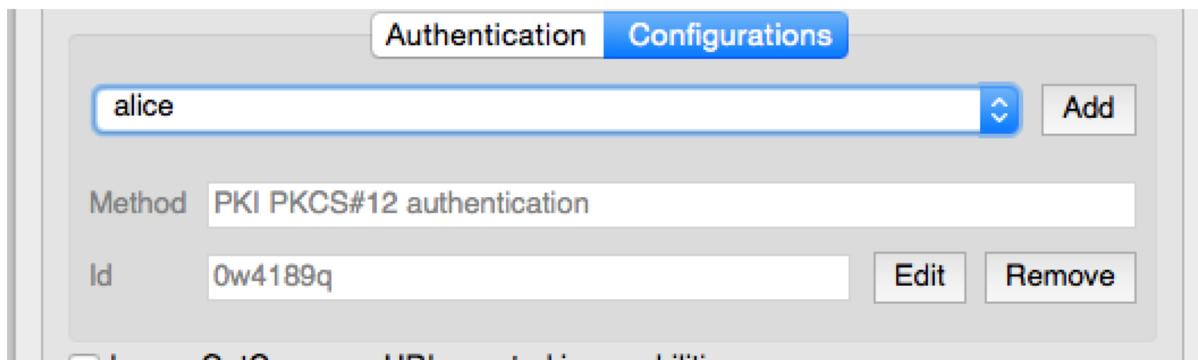
14.5.1 GUI om persoonlijke gegevens te selecteren

If it's necessary to select a *Authentication Configuration* from the set stored in the *Authentication DB* it is available in the GUI class QgsAuthConfigSelect

en kan worden gebruikt zoals in het volgende snippet:

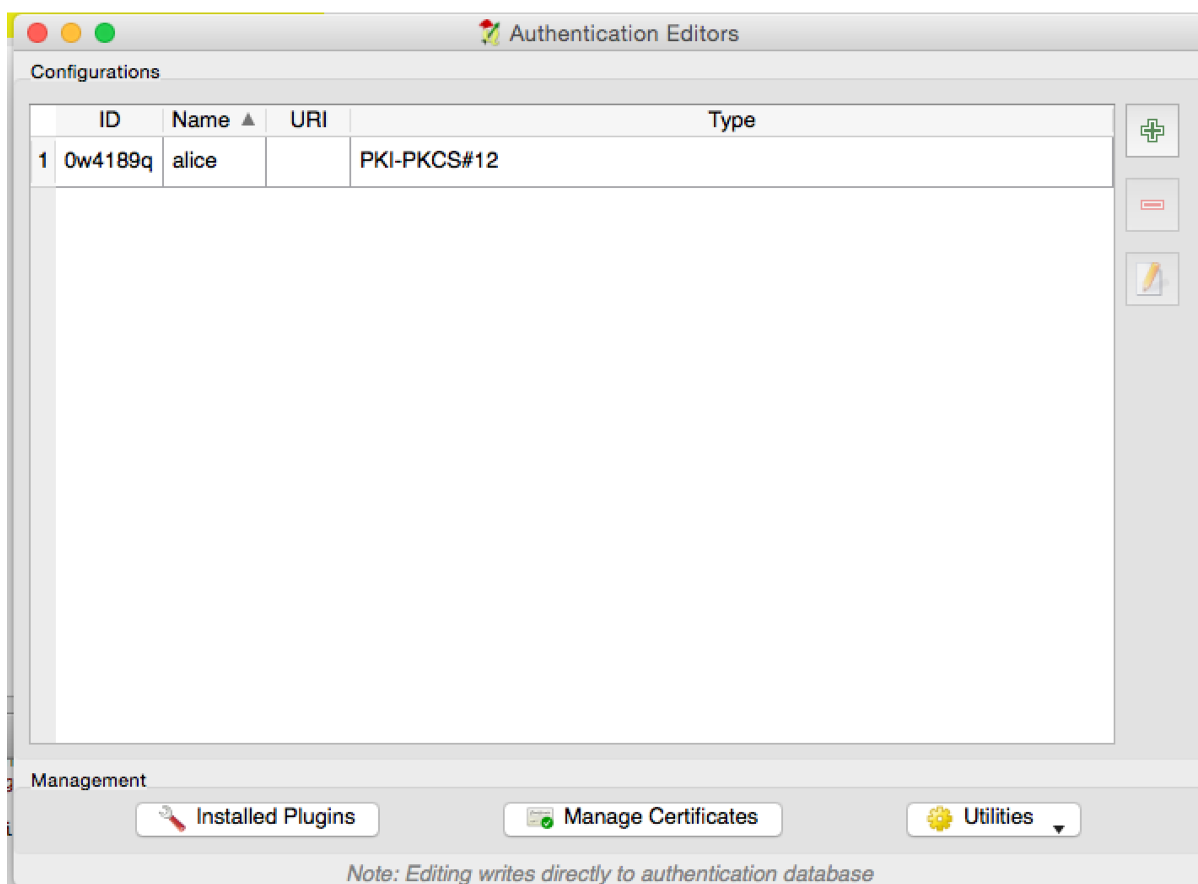
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
# GUI has to be integrated
tabGui.insertTab( 1, gui, "Configurations" )
```

The above example is get from the QGIS source code The second parameter of the GUI constructor refers to data provider type. The parameter is used to restrict the compatible *Authentication Methods* with the specified provider.



14.5.2 Bewerkers voor GUI authenticatie

The complete GUI used to manage credentials, authorities and to access to Authentication utilities is managed by the class QgsAuthEditorWidgets



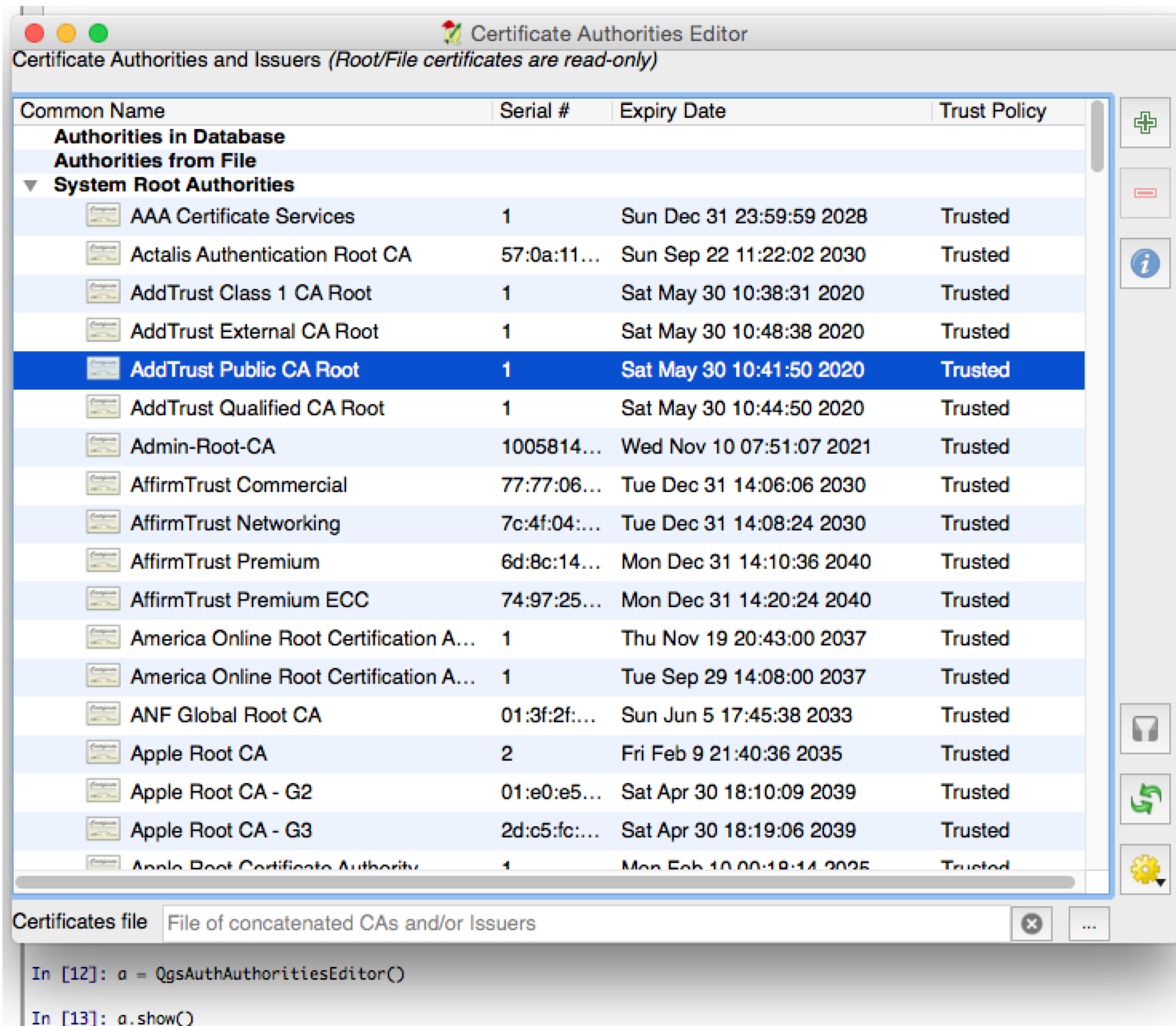
en kan worden gebruikt zoals in het volgende snippet:

```
# create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent )
gui.show()
```

een geïntegreerd voorbeeld is te vinden in de gerelateerde test

14.5.3 Bewerker voor GUI autoriteiten

A GUI used to manage only authorities is managed by the class QgsAuthAuthoritiesEditor


Certificate Authorities Editor
Certificate Authorities and Issuers (Root/File certificates are read-only)
Common Name
Serial #
Expiry Date
Trust Policy
Authorities in Database
Authorities from File
System Root Authorities
AAA Certificate Services 1 Sun Dec 31 23:59:59 2028 Trusted
Actalis Authentication Root CA 57:0a:11… Sun Sep 22 11:22:02 2030 Trusted
AddTrust Class 1 CA Root 1 Sat May 30 10:38:31 2020 Trusted
AddTrust External CA Root 1 Sat May 30 10:48:38 2020 Trusted
AddTrust Public CA Root 1 Sat May 30 10:41:50 2020 Trusted
AddTrust Qualified CA Root 1 Sat May 30 10:44:50 2020 Trusted
Admin-Root-CA 1005814… Wed Nov 10 07:51:07 2021 Trusted
AffirmTrust Commercial 77:77:06… Tue Dec 31 14:06:06 2030 Trusted
AffirmTrust Networking 7c:4f:04:… Tue Dec 31 14:08:24 2030 Trusted
AffirmTrust Premium 6d:8c:14… Mon Dec 31 14:10:36 2040 Trusted
AffirmTrust Premium ECC 74:97:25… Mon Dec 31 14:20:24 2040 Trusted
America Online Root Certification A… 1 Thu Nov 19 20:43:00 2037 Trusted
America Online Root Certification A… 1 Tue Sep 29 14:08:00 2037 Trusted
ANF Global Root CA 01:3f:2f:… Sun Jun 5 17:45:38 2033 Trusted
Apple Root CA 2 Fri Feb 9 21:40:36 2035 Trusted
Apple Root CA - G2 01:e0:e5… Sat Apr 30 18:10:09 2039 Trusted
Apple Root CA - G3 2d:c5:fc:… Sat Apr 30 18:19:06 2039 Trusted
Certificates file
File of concatenated CAs and/or Issuers
In [12]: a = QgsAuthAuthoritiesEditor()
In [13]: a.show()

en kan worden gebruikt zoals in het volgende snippet:

```
# create the instance of the QgsAuthAuthoritiesEditor GUI hierarchically
#  linked to the widget referred with 'parent'
gui = QgsAuthAuthoritiesEditor( parent )
gui.show()
```

CHAPTER 15

Instellingen voor de IDE voor het schrijven en debuggen van plug-ins

- Een opmerking voor het configureren van uw IDE op Windows
- Debuggen met behulp van Eclipse en PyDev
  - Installatie
  - QGIS voorbereiden
  - Eclipse instellen
  - Configureren van de debugger
  - Eclipse de API laten begrijpen
- Debuggen met behulp van PDB

Hoewel elke programmeur zijn eigen voorkeur heeft voor een IDE/tekstbewerker, zijn hier enkele aanbevelingen voor het instellen van enkele populaire IDE's voor het schrijven en debuggen van plug-ins voor Python in QGIS.

15.1 Een opmerking voor het configureren van uw IDE op Windows

Op Linux is geen aanvullende configuratie nodig om plug-ins te ontwikkelen. Maar op Windows dient u er voor te zorgen dat u dezelfde instellingen voor de omgeving heeft en dezelfde bibliotheken en interpreter gebruikt als QGIS. De snelste manier om dit te doen is om het opstartbestand van QGIS aan te passen.

Als u het installatieprogramma van OSGeo4W gebruikte, vindt u dit in de map `bin` van uw installatie van OSGeoW. Zoek naar iets als `C:\OSGeo4W\bin\qgis-unstable.bat`.

Voor het gebruiken van Pyscripter IDE, is dit wat u moet doen:

- Maak een kopie van `qgis-unstable.bat` en hernoem die naar `pyscripter.bat`.
- Open het in een bewerker. En verwijder de laatste regel, die welke QGIS laat starten.
- Voeg een regel toe die verwijst naar uw uitvoerbare bestand van Pyscripter en voeg het argument voor de opdrachtregel toe dat de te gebruiken versie van Python instelt (2.7 in het geval van QGIS >= 2.0)
- Voeg ook het argument toe dat verwijst naar de map waar Pyscripter de Python dll kan vinden die wordt gebruikt door QGIS, u vindt deze in de map bin van uw installatie van 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
  ```

Wanneer u nu dubbelklikt op dit batch-bestand, zal dat Pyscripter starten, met het juiste pad.

Meer populair dan Pyscripter, is Eclipse een veel voorkomende keuze bij ontwikkelaars. In de volgende gedeelten zullen we uitleggen hoe het te configureren voor het ontwikkelen en testen van plug-ins. U zou ook een batch-bestand moeten maken en dat gebruiken om Eclipse te starten om uw omgeving voor te bereiden om Eclipse in Windows te gebruiken.

Volg deze stappen om het batch-bestand te maken:

- Zoek naar de map waar het bestand file:*qgis_core.dll* is geplaatst. Normaal gesproken is dit `C:\OSGeo4W\apps\qgis\bin`, maar als u uw eigen toepassing in QGIS compileerde is het in de map waar u het bouwde in `output/bin/RelWithDebInfo`
- Zoek naar uw uitvoerbare bestand `eclipse.exe`.
- Maak het volgende script en gebruik dat om Eclipse te starten bij het ontwikkelen van plug-ins voor 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
```

15.2 Debuggen met behulp van Eclipse en PyDev

15.2.1 Installatie

Zorg er voor, om Eclipse te kunnen gebruiken, dat u het volgende heeft geïnstalleerd

- Eclipse
- Aptana Eclipse Plugin of PyDev
- QGIS 2.x

15.2.2 QGIS voorbereiden

Er moet enige voorbereiding worden gedaan in QGIS zelf. Twee plug-ins zijn van belang: **Remote Debug** en **Plugin reloader**.

- Ga naar *Plug-ins* → *Plug-ins beheren en installeren*
- Zoek naar **Remote Debug** (op dit moment is die nog steeds experimenteel, dus schakel Experimentele plug-ins in onder de tab :guilabel:'Opties'in het geval hij niet wordt weergegeven). Installeer het.
- Zoek naar *Plugin reloader* en installeer die ook. Dit stelt u in staat een plug-in opnieuw op te starten in plaats va die te moeten sluiten en QGIS opnieuw op te moeten starten om hem opnieuw te laden.

15.2.3 Eclipse instellen

Maak, in Eclipse, een nieuw project. U kunt *General Project* selecteren en uw echte bronnen later koppelen, dus het maakt niet echt uit waar u dit project plaatst.

Klik nu met rechts op uw nieuwe project en kies *New* → *Folder*.

Klik op **[Advanced]** en kies *Link to alternate location (Linked Folder)*. In het geval dat u al bronnen heeft die u wilt debuggen, kies die, in het geval u die niet heeft, maak een map aan zoals al eerder is uitgelegd

Nu zal in de weergave *Project Explorer* uw boom van bronnen opkomen en kunt u beginnen met het werken aan de code. U heeft al accentuering van syntaxis en alle andere krachtige gereedschappen voor de IDE beschikbaar.

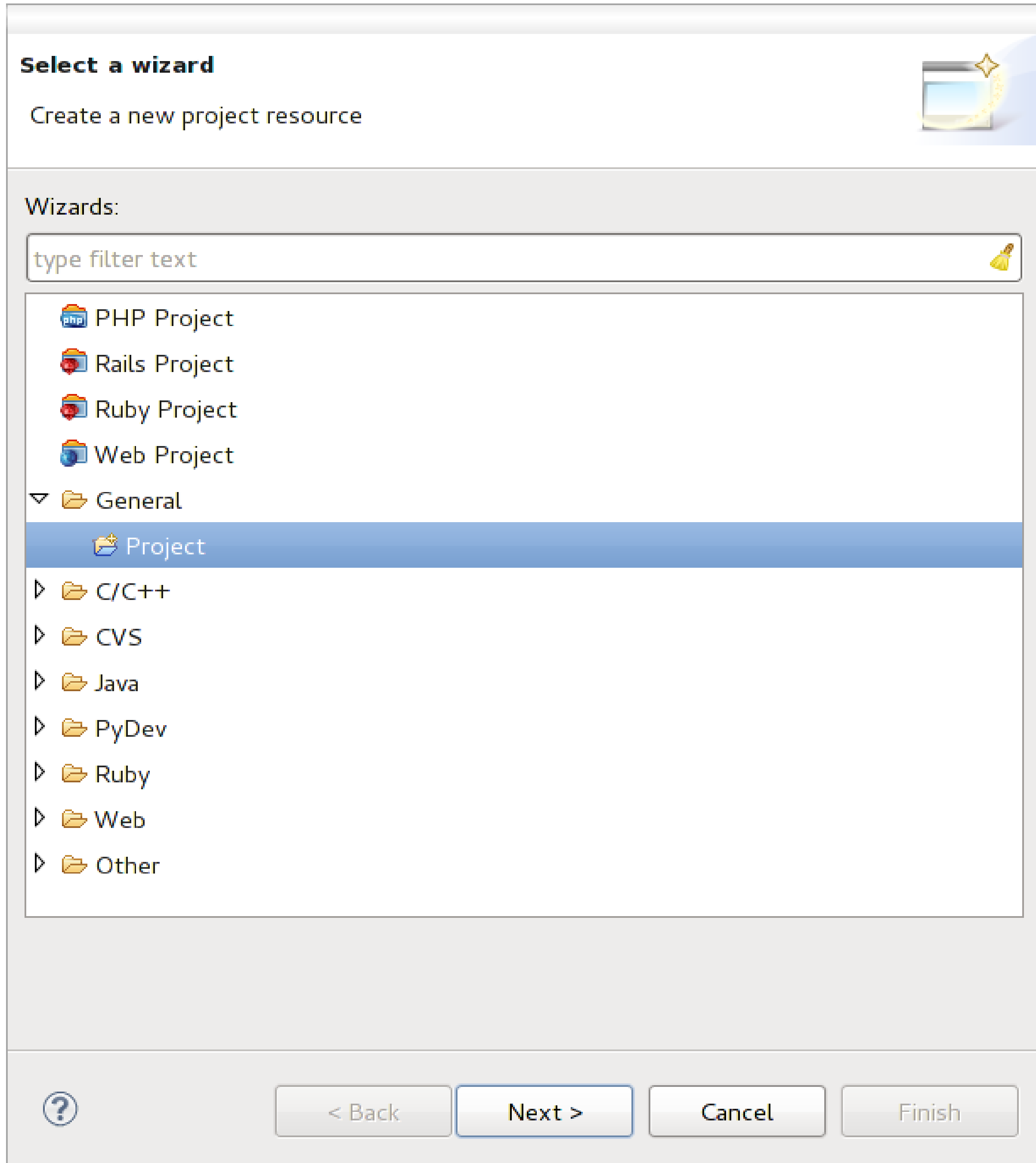


Figure 15.1: Eclipse-project

15.2.4 Configureren van de debugger

Schakel naar het perspectief Debug in Eclipse (*Window* → *Open Perspective* → *Other* → *Debug*) om de debugger werkend te krijgen.

Start nu de server voor debuggen van PyDev door te kiezen *PyDev* → *Start Debug Server*.

Eclipse wacht nu op een verbinding vanuit QGIS naar zijn server voor debuggen en wanneer QGIS verbindt met de server voor debuggen zal dat het mogelijk maken de scripts van Python te beheren. Dat is dus precies waarom we de plug-in *Remote Debug* hebben geïnstalleerd. Dus start QGIS voor het geval u dat nog niet gedaan heeft en klik op het symbool Bug.

Nu kunt u een onderbrekingspunt instellen en zodra als dat wordt tegengekomen door de code, zal de uitvoering stoppen en kunt u de huidige status van uw plug-in inspecteren. (Het onderbrekingspunt is de groene punt in de afbeelding hieronder, stel er een in door dubbel te klikken in de witte ruimte links van de regel waarvoor u wilt dat het onderbrekingspunt wordt ingesteld).

```
 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 15.2: Onderbrekingspunt

Een zeer interessant ding waarvan u nu gebruik kunt maken is de console voor debuggen. Zorg er voor dat de uitvoering nu wordt gestopt op een onderbrekingspunt, voordat u doorgaat.

Open de weergave van de Console (*Window* → *Show view*). Het zal de console *Debug Server* weergeven die niet bijzonder interessant is. Maar er is een knop **[Open Console]** die u brengt naar een meer interessante PyDev Debug Console. Klik op de pijl naast de knop **[Open Console]** en kies *PyDev Console*. Een venster opent om u te vragen welke console u wilt starten. Kies *PyDev Debug Console*. In het geval dat die is uitgegrijsd en u zegt om de debugger te starten en het geldige frame te selecteren, zorg er dan voor dat u de debugger op afstand heeft aangekoppeld en momenteel op een onderbrekingspunt staat.

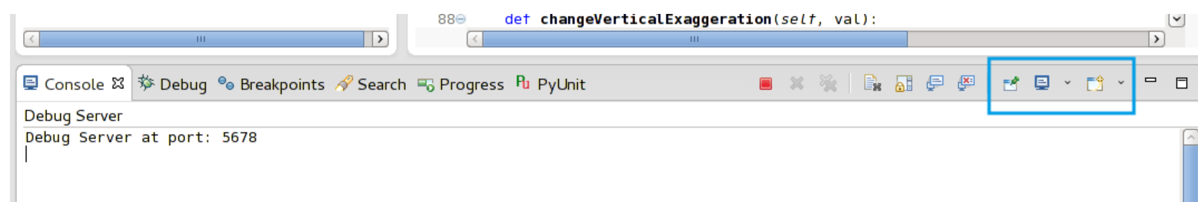


Figure 15.3: PyDev console voor debuggen

U heeft nu een interactieve console die u opdrachten laat testen vanuit de huidige context. U kunt variabelen manipuleren of aanroepen naar de API maken of wat u ook wilt.

Enigszins vervelend is dat, elke keer als u een opdracht invoert, de console terugschakelt naar de Debug Server. U kunt op de knop *Pin Console* klikken als u op de pagina van de Debug Server bent en het zou deze beslissing, ten minste voor de huidige sessie van debuggen, moeten onthouden om dit gedrag te stoppen,

15.2.5 Eclipse de API laten begrijpen

Een zeer handige mogelijkheid is om Eclipse kennis te laten nemen van de API van QGIS. Dit stelt u in staat om het uw code te laten controleren op typefouten. Maar niet alleen dat, het stelt Eclipse ook in staat om u te helpen met automatisch aanvullen vanuit het importeren naar aanroepen van de API.

Eclipse parst de bibliotheekbestanden van QGIS en krijgt daar vandaan alle informatie om dit te doen. Het enige dat u moet doen is Eclipse vertellen waar het de bibliotheken kan vinden.

Klik op *Window* → *Preferences* → *PyDev* → *Interpreter* → *Python*.

U zult uw geconfigureerde interpreter voor Python zien in het bovenste gedeelte van het venster (op dit moment Python2.7 voor QGIS) en enkele tabs in het onderste gedeelte. De voor ons interessante tabs zijn *Libraries* en *Forced Builtins*.

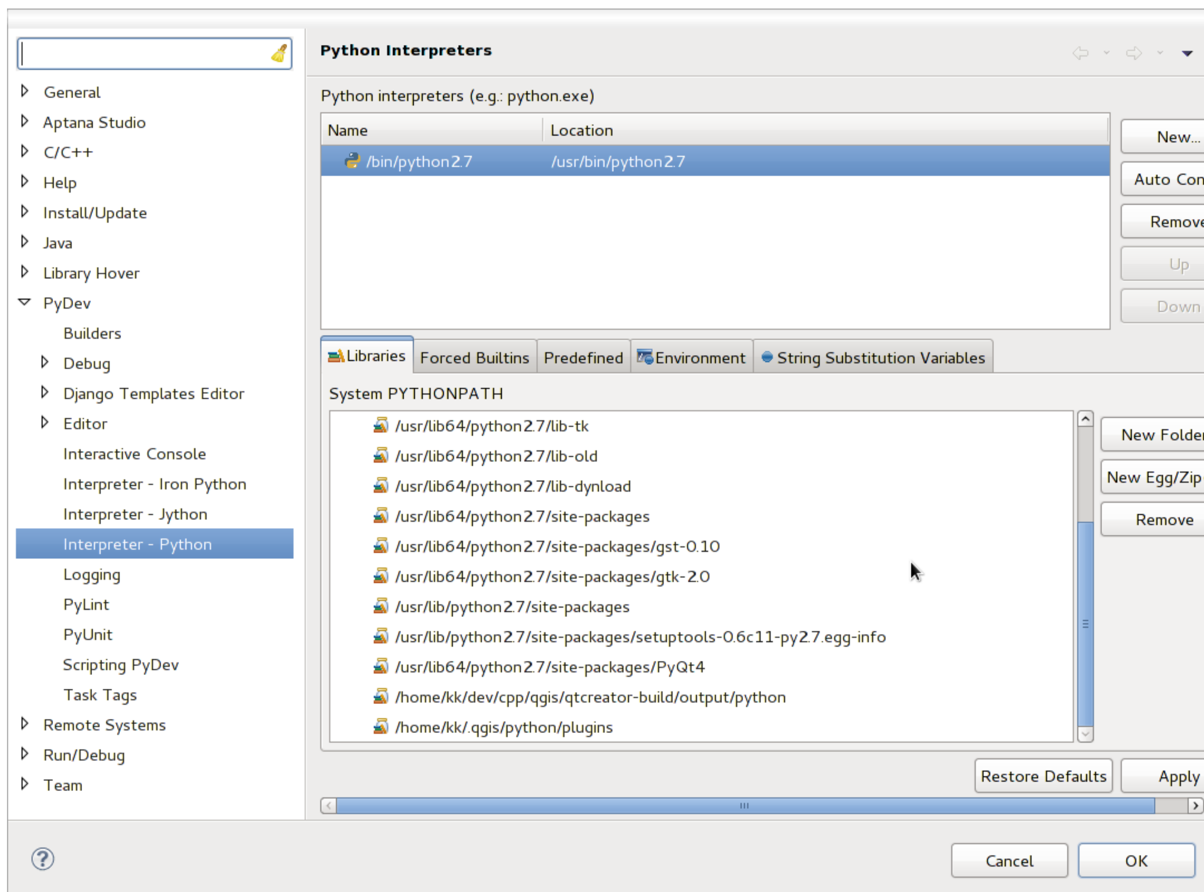


Figure 15.4: PyDev console voor debuggen

Open eerst de tab Libraries. Voeg een nieuwe map toe en kies de map voor Python van uw installatie voor QGIS. Als u niet weet waar die map staat (hij staat niet in de map plug-ins) open QGIS, start een console voor Python en voer eenvoudigweg `qgis` in en druk op Enter. Het zal u tonen welke module QGIS gebruikt en het pad er van. Verwijder het achterliggende `/qgis/__init__.pyc` uit dit pad en u heeft het pad waar u naar zoekt.

U zou hier ook uw map voor plug-ins moeten toevoegen (op Linux is dat `~/.qgis2/python/plugins`).

Spring vervolgens naar de tab *Forced Builtins*, klik op *New...* en voer in `qgis`. Dit zal Eclipse de API van QGIS laten parsen. U wilt waarschijnlijk ook dat Eclipse weet heeft van de API voor PyQt4. Voeg daarom ook PyQt4 toe als forced builtin. Die zou waarschijnlijk al aanwezig zijn op uw tab Libraries.

Klik op *OK* en u bent klaar.

Notitie: Elke keer dat de API van QGIS API wijzigt (bijv. als u de master van QGIS compileert en het bestand

SIP wijzigt), zou u terug moeten gaan naar deze pagina en eenvoudigweg op *Apply* moeten klikken. Dat laat Eclipse alle bibliotheken opnieuw parsen.

15.3 Debuggen met behulp van PDB

Als u geen IDE gebruikt, zoals Eclipse, kunt u debuggen met behulp van PDB, volg deze stappen.

Voeg eerst de code toe op de plaats waar u wilt debuggen

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

Voer dan QGIS uit vanaf de opdrachtregel.

Doe op Linux:

```
$ ./Qgis
```

Doe op MacOS:

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

En wanneer de toepassing uw onderbrekingspunt tegenkomt kunt u in de console typen!

TODO: Informatie voor testen toevoegen

CHAPTER 16

Plug-in-lagen gebruiken

Als uw plug-in zijn eigen methoden gebruikt om een kaartlaag te renderen, zou het schrijven van uw eigen laagtype, gebaseerd op QgsPluginLayer, de beste manier kunnen zijn on dat te implementeren.

TODO: Juistheid controleren en uitgebreider goed te gebruiken gevallen voor QgsPluginLayer weergeven, ...

16.1 Sub-klassen in QgsPluginLayer

Hieronder staat een voorbeeld van een minimale implementatie van QgsPluginLayer. Het is een uittreksel van de voorbeeld plug-in Watermark

```
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
```

Methoden voor het lezen en schrijven van specifieke informatie naar het projectbestand kan ook worden toegevoegd

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

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

Bij het laden van een project dat een dergelijke laag bevat, is een klasse factory nodig

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

U kunt ook code toevoegen voor het weergeven van aangepaste informatie in de eigenschappen van de laag

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

CHAPTER 17

Compatibiliteit met oudere versies van QGIS

17.1 Menu Plug-ins

Als u uw menuitems voor de plug-in plaatst in één van de nieuwe menu's (*Raster*, *Vector*, *Database* of *Web*), zou u de code van de functies `initGui()` en `unload()` moeten aanpassen. Omdat deze menu's alleen beschikbaar zijn in QGIS 2.0 en hoger, is de eerste stap om te controleren of de uitgevoerde versie van QGIS alle benodigde functies heeft. Als de nieuwe menu's beschikbaar zijn, zullen we onze plug-in onder dit menu plaatsen, anders zullen we het oude menu *Plug-ins* gebruiken. Hier is een voorbeeld voor het menu *Raster*

```
def initGui(self):
  # create action that will start plugin configuration
  self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWindo
  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.renderTes
```

CHAPTER 18

Uw plug-in uitgeven

- Metadata en namen
- Code en hulp
- Officiële Python plug-in opslagplaats
  - Rechten
  - Beheer van 'trust'
  - Validatie
  - Plug-in structuur

Als uw plug-in eenmaal klaar is en u denkt dat de plug-in van nut zou kunnen zijn voor anderen, aarzel dan niet om het te uploaden naar *Officiële Python plug-in opslagplaats*. Op die pagina kunt u ook richtlijnen vinden voor het verpakken om de plug-in voor te bereiden om goed te werken met het installatieprogramma van plug-ins. Of, in het geval u uw eigen opslagplaats voor plug-ins zou willen inrichten, maak een eenvoudig XML-bestand dat de plug-ins en hun metadata vermeld, voor voorbeelden zie andere opslagplaatsen voor plug-ins.

Neem goede notie van de volgende suggesties:

18.1 Metadata en namen

- vermijd het gebruiken van een naam die teveel lijkt op die van bestaande plug-ins
- als uw plug-in een soortgelijke functionaliteit heeft als een bestaande plug-in, leg dan de verschillen uit in het vak About, zodat de gebruiker weet welke te gebruiken zonder dat hij hem eerst moet installeren en testen
- vermijd het herhalen van "plug-in" in de naam van de plug-in zelf
- gebruik het veld Description in de metadata voor een omschrijving van één regel, het veld About voor meer gedetailleerde instructies
- neem een code repository, een bug tracker, en een homepage op; dat zal de mogelijkheid tot samenwerken enorm vergroten, en kan zeer eenvoudig worden gedaan met behulp van één van de beschikbare infrastructuren voor het web (GitHub, GitLab, Bitbucket, etc.)
- kies tags met zorg: vermijd de niet-informatieve (bijv. vector) en gebruik bij voorkeur die welke al zijn gebruikt door anderen (bekijk de website van de plug-in)
- voeg een juist pictogram toe, volsta niet met het standaard pictogram; bekijk de interface van QGIS voor een suggestie van de te gebruiken stijl

18.2 Code en hulp

- neem geen gegenereerd bestand (ui_*.py, resources_rc.py, gegenereerde Helpbestanden…) op en waardeloos spul (bijv. .gitignore) in de repository
- voeg de plug-in toe aan het toepasselijke menu (Vector, Raster, Web, Database)
- indien van toepassing (plug-ins die analyses uitvoeren), overweeg dan om de plug-in toe te voegen als een subplug-in voor het framework Processing: dat zal het voor gebruikers mogelijk maken het in batch uit te voeren, het te integreren in meer complexe werkstromen, en zal u bevrijden van het ontwerpen van een interface
- neem tenminste minimale documentatie op en, indien nuttig voor testen en begrijpen, voorbeeldgegevens.

18.3 Officiële Python plug-in opslagplaats

U vindt de *officiële* Python plug-in opslagplaats op http://plugins.qgis.org/.

Voor het gebruiken van de officiële opslagplaats moet u een OSGEO ID verkrijgen van het OSGEO webportaal.

Als u uw plug-in eenmaal heeft geüpload, zal die worden gekeurd door een lid van de staf en zult u bericht ontvangen.

TODO: Een koppeling naar het document voor governance invoegen

18.3.1 Rechten

Deze regels zijn geïmplementeerd in de officiële plug-in opslagplaats:

- elke geregistreerde gebruiken mag een nieuwe plug-in toevoegen
- *staf*-gebruikers mogen alle versies van plug-ins goed- of afkeuren
- gebruikers die het speciale recht *plugins.can_approve* hebben krijgen de versies die zij uploaden automatisch goedgekeurd
- gebruikers die het speciale recht *plugins.can_approve* hebben kunnen door anderen geüploade versies goedkeuren zo lang als zij in de lijst plug-in *owners* staan
- een bepaalde plug-in kan worden verwijderd en bewerkt, alleen door *staf*-gebruikers en plug-in *owners*
- als een gebruiker zonder het recht *plugins.can_approve* een nieuwe versie uploadt, wordt de versie van de plug-in automatisch niet goedgekeurd.

18.3.2 Beheer van 'trust'

Stafleden kunnen het recht *trust* toekennen aan geselecteerde makers van plug-ins door het instellen van het recht *plugins.can_approve* door middel van de front-end toepassing.

De gedetailleerde weergave van plug-ins biedt directe koppelingen om trust toe te kennen aan de maker van de plug-in of de plug-in *owners*.

18.3.3 Validatie

Metadata van plug-ins worden automatisch geïmporteerd en gevalideerd vanuit het gecomprimeerde pakket als de plug-in wordt geüpload.

Hier zijn enkele regels voor validatie waarvan u op de hoogte zou moeten zijn wanneer u een plug-in zou willen uploaden naar de officiële opslagplaats:

1. de naam van de hoofdmap die uw plug-in bevat mag alleen ASCII-tekens bevatten (A-Z en a-z), cijfers en het teken underscore (_) en minus (-), ook mag het niet beginnen met een cijfer
2. `metadata.txt` is vereist
3. alle vereiste metadata vermeld in *metadata table* moeten aanwezig zijn
4. het veld *version* voor metadata moet uniek zijn

18.3.4 Plug-in structuur

Op grond van de regels voor validatie moet het gecomprimeerde (.zip) pakket van uw plug-in een specifieke structuur hebben om als een functionele plug-in te worden gevalideerd. Omdat de plug-in zal worden uitgepakt binnen de map plug-ins van de gebruiker moet het zijn eigen map binnen het .zip-bestand hebben om niet te interfereren met andere plug-ins. Verplichte bestanden zijn: `metadata.txt` en `__init__.py`. Maar het zou leuk zijn om een `README` te hebben en natuurlijk een pictogram om de plug-in weer te geven (`resources.qrc`). Hieronder volgt een voorbeeld van hoe een plug-in.zip er uit zou moeten zien.

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

CHAPTER 19

Codesnippers

- Hoe een methode aan te roepen met een sneltoets
- Hoe te schakelen tussen lagen
- Hoe toegang te krijgen tot de attributentabel van geselecteerde objecten

Dit gedeelte behandelt codesnippers om de ontwikkeling van plug-ins te faciliteren.

19.1 Hoe een methode aan te roepen met een sneltoets

Voeg in de plug-in de `initGui()` toe

```
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)
```

Aan `unload()` voeg toe

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

De methode die wordt aangeroepen wanneer op F7 wordt gedrukt

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

19.2 Hoe te schakelen tussen lagen

Vanaf QGIS 2.4 is er een nieuwe API voor de lagenboom die directe toegang tot de lagenboom in de legenda toestaat. Hier is een voorbeeld om te schakelen met de zichtbaarheid van de actieve laag

```
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)
```

19.3 Hoe toegang te krijgen tot de attributentabel van geselecteerde objecten

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

De methode vereist één parameter (de nieuwe waarde voor de attribuutveld van het geselecteerde object(en)) en kan worden aangeroepen met

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

CHAPTER 20

Een plug-in voor Processing schrijven

- Creating a plugin that adds an algorithm provider
- Creating a plugin that contains a set of processing scripts

Afhankelijk van het soort plug-in dat u gaat ontwikkelen, zou het misschien een betere optie zijn om de functionaliteit ervan toe te voegen als een algortime voor Processing (of een set daarvan). Dat zou tot een betere integratie in QGIS leiden, aanvullende functionaliteit (omdat het kan worden uitgevoerd in de componenten van Processing, zoals Grafische modellen bouwen of de Processing interface voor batchverwerking), en een snellere ontwikkelingstijd (omdat Processing een groot deel van het werk zal overnemen).

This document describes how to create a new plugin that adds its functionality as Processing algorithms.

There are two main mechanisms for doing that:

- Creating a plugin that adds an algorithm provider: This options is more complex, but provides more flexibility
- Creating a plugin that contains a set of processing scripts: The simplest solution, you just need a set of Processing script files.

20.1 Creating a plugin that adds an algorithm provider

To create an algorithm provider, follow these steps:

- Installeer de plug-in Plugin Builder
- Maak een nieuwe plug-in met de Plugin Builder. Wanneer de Plugin Builder u vraagt naar het te gebruiken sjabloon, selecteer dan "Processing provider".
- De gemaakte plug-in bevat een provider met één enkel algoritme. Zowel het bestand voor de provider als het bestand voor het algoritme zijn volledig voorzien van commentaar en bevatten informatie over hoe de provider aan te passen en aanvullende algoritmen toe te voegen. Verwijs daarnaar voor meer meer informatie.

20.2 Creating a plugin that contains a set of processing scripts

To create a set of processing scripts, follow these steps:

- Create your scripts as described in the PyQGIS cookbook. All the scripts that you want to add, you should have them available in the Processing toolbox.
- In the *Scripts/Tools* group in the Processing toolbox, double-click on the *Create script collection plugin* item. You will see a window where you should select the scripts to add to the plugin (from the set of available ones in the toolbox), and some additional information needed for the plugin metadata.

- Click on OK and the plugin will be created.
- You can add additional scripts to the plugin by adding scripts python files to the *scripts* folder in the resulting plugin folder.

CHAPTER 21

Bibliotheek Netwerkanalyse

- Algemene informatie
- Een grafiek bouwen
- Grafiekanalyse
  - Kortste pad zoeken
  - Beschikbare gebieden

Beginnend vanaf revisie ee19294562 (QGIS >= 1.8) werd de nieuwe bibliotheek Network analysis toegevoegd aan bron-analysebibliotheek van QGIS. De bibliotheek:

- maakt rekenkundige grafieken uit geografische gegevens (polylijn vectorlagen)
- implementeert basismethoden vanuit grafiektheorie (momenteel alleen Dijkstra's algoritme)

De bibliotheek Network analysis werd gemaakt door het exporteren van basisfuncties vanuit de bronplug-in RoadGraph en nu kunt u de methoden daarvan in plug-ins gebruiken of direct vanuit de console voor Python.

21.1 Algemene informatie

In het kort kan een typisch gebruik worden omschreven als:

1. maakt rekenkundige grafiek uit geo-gegevens (gewoonlijk polylijn vectorlaag)
2. voert grafiekanalyse uit
3. gebruikt resultaten van analyse (door ze, bijvoorbeeld, te visualiseren)

21.2 Een grafiek bouwen

Het eerste dat u moet doen — is om invoergegevens voor te bereiden, dat is een vectorlaag converteren naar een grafiek. Alle verdere acties zullen deze grafiek gebruiken, niet de laag.

Als een bron kunnen we elke polylijn vectorlaag gebruiken. Knopen van de polylijnen worden punten in de grafiek, en segmenten van de polylijnen worden randen van de grafiek. Indien verscheidene knopen dezelfde coördinaten hebben dan zijn zij dezelfde knop in de grafiek. Dus twee lijnen die een gemeenschappelijk knoop hebben worden aan elkaar verbonden.

Aanvullend, gedurende het maken van de grafiek is het mogelijk om de een willekeurige aantal aanvullende punten "vast te zetten" ("te verbinden") aan de invoer vectorlaag. Voor elk aanvullend punt zal een overeenkomst worden gevonden — het dichtstbijzijnde punt in de grafiek of de dichtstbijzijnde gelegen rand. In het laatste geval zal de rand worden gesplitst en een nieuw punt worden toegevoegd.

Attributen van de vectorlaag en de lengte van een rand kunnen worden gebruikt als de eigenschappen van een rand.

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

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

Het is tijd om het proces in te duiken.

Als eerste, om deze bibliotheek te kunnen gebruiken, zouden we de module Network analysis moeten importeren

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

Dan enkele voorbeelden voor het maken van een 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)
```

We zouden, om een director te construeren, een vectorlaag door moeten geven, die zal worden gebruikt als de bron voor de structuur van de grafiek en informatie over toegestane bewegingen over elke segment van de weg (één richting of beide, directe of tegengestelde richting). De aanroep ziet er uit zoals deze

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

En hier is een volledige lijst van wat deze parameters betekenen:

- `vl` — vectorlaag gebruikt om de grafiek te bouwen
- `directionFieldId` — index van het attribuut tabelveld, waar informatie over de richting van de wegen is opgeslagen. Indien `-1`, gebruik deze informatie dan helemaal niet. Een integer.
- `directDirectionValue` — veldwaarde voor wegen met een directe richting (verplaatsen vanaf het eerste punt op de lijn tot het laatste). Een string.
- `reverseDirectionValue` — veldwaarde voor wegen met een tegengestelde richting (verplaatsen vanaf het laatste punt op de lijn tot het eerste). Een string.
- `bothDirectionValue` — veldwaarde voor wegen in beide richtingen (voor dergelijke wegen kunnen we verplaatsen van het eerste punt naar het laatste en van laatste naar eerste). Een string.
- `defaultDirection` — standaard richting van de weg. Deze waarde zal worden gebruikt voor die wegen waar het veld `directionFieldId` niet is ingesteld of ene andere waarde heeft dan een van de hierboven gespecificeerde drie waarden. Een integer. `1` geeft de directe richting aan, `2` geeft de tegengestelde richting aan en `3` geeft beide richtingen aan.

Het is dan nodig om een strategie te maken voor het berekenen van de eigenschappen van de rand

```
properter = QgsDistanceArcProperter()
```

En de director vertellen over deze strategie

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

Nu kunnen we de builder gebruiken, wat de grafiek zal maken. De klasseconstructor QgsGraphBuilder kan verschillende argumenten aannemen:

- crs — te gebruiken coördinaten referentiesysteem. Verplicht argument.
- otfEnabled — opnieuw projecteren met "Gelijktijdige CRS-transformatie gebruiken" gebruiken of niet. Standaard const:*True* (gebruik OTF).
- topologyTolerance — topologische tolerantie. Standaard waarde is 0.
- ellipsoidID — te gebruiken ellipsoïde. Standaard "WGS84".

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

Ook kunnen we verscheidene punten definiëren, die zullen worden gebruikt in de analyse. Bijvoorbeeld

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

Nu is alles op zijn plaats dus kunnen we de grafiek bouwen en deze punten daaraan "verbinden"

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

Bouwen van de grafiek kan enige tijd vergen (wat afhankelijk is van het aantal objecten in een laag en de grootte van de laag). `tiedPoints` is een lijst met coördinaten van de "verbonden" punten. Als de bewerking van het bouwen is voltooid kunnen we de grafiek nemen en die gebruiken voor de analyse

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

Met de volgende code kunnen we de vertex-indexen verkrijgen van onze punten

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

21.3 Grafiekanalyse

Netwerkanalyse wordt gebruikt om antwoord te vinden op twee vragen: welke punten zijn verbonden en hoe het kortste pad te vinden. De bibliotheek Network analysis verschaft Dijkstra's algoritme om deze problemen op te lossen.

Dijkstra's algoritme zoekt de kortste route van één van de punten van de grafiek naar alle andere en de waarden van de parameters voor optimalisatie. De resultaten kunnen worden weergegeven als een kortste pad-boom.

De kortste pad-boom is een gedirigeerde gewogen grafiek (of meer precies — boom) met de volgende eigenschappen:

- slechts één punt heeft geen inkomende randen — de wortel van de boom
- alle andere punten hebben slechts één inkomende rand
- als punt B bereikbaar is vanuit punt A, dan is het pad van A naar B het enige beschikbare pad en is het optimaal (kortste) op deze grafiek

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

De methode `shortestTree()` is handig wanneer u over de boom van het kortste pad wilt wandelen. Het maakt altijd een nieuw grafiekobject (QgsGraph) en accepteert drie variabelen:

- source — grafiek voor invoer

- startVertexIdx — index van het punt op de boom (de wortel van de boom)
- criterionNum — nummer van te gebruiken eigenschap van de rand (beginnend vanaf 0).

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

De methode `dijkstra()` heeft dezelfde argumenten, maar geeft twee arrays terug. In het eerste array bevat element i de index van de inkomende rand of -1 als er gene inkomende randen zijn. In de tweede array bevat element i de afstand van de wortel van de boom tot het punt i of DOUBLE_MAX als het punt i onbereikbaar is vanuit de wortel.

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

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

```
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
```

Hetzelfde maar met behulp van de methode `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())
```

21.3.1 Kortste pad zoeken

De volgende benadering wordt gebruikt om het optimale pad tussen twee punten te zoeken. Beide punten (begin A en einde B) zijn "verbonden" met de grafiek wanneer die wordt gebouwd. Met behulp van de methoden `shortestTree()` of `dijkstra()` bouwen we dan de boom voor het kortste pad met de wortel in beginpunt A. In dezelfde boom zoeken we ook naar eindpunt B en beginnen te lopen door de boom vanaf punt B naar punt A. Het gehele algoritme kan worden geschreven als

```
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
```

Op dit punt hebben we het pad, in de vorm van de geïnverteerde lijst van punten (punten zijn vermeld in de omgekeerde volgorde van eindpunt naar beginpunt) die zullen worden bezocht gedurende het lopen over dit pad.

Hier is de voorbeeldcode voor de console van Python in QGIS (u dient een lijnstring laag te selecteren in de inhoudsopgave en de coördinaten te vervangen door uw eigen) dat de methode `shortestTree()` gebruikt

```
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)
```

En hier is hetzelfde voorbeeld, maar met behulp van de methode `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)
```

21.3.2 Beschikbare gebieden

Het beschikbare gebied voor punt A is de subset van punten op de grafiek die toegankelijk zijn vanuit punt A en de kosten van de paden van A naar deze punten zijn niet groter dan een bepaalde waarde.

Dit kan duidelijker worden weergegeven met behulp van het volgende voorbeeld: "Er is een brandweergarage. Welke delen van de stad kan een brandweerauto bereiken in 5 minuten? 10 minuten? 15 minuten?". De antwoorden op deze vragen zijn de beschikbare gebieden voor deze brandweergarage.

We kunnen de methode `dijkstra()` van de klasse `QgsGraphAnalyzer` gebruiken om de beschikbare gebieden te zoeken. Het is voldoende om de elementen van de array met kosten te vergelijken met een vooraf gedefinieerde waarde. Als de kosten[i] minder zijn dan of gelijk zijn aan een vooraf gedefinieerde waarde, dan ligt punt i binnen het beschikbare gebied, anders ligt het er buiten.

Een wat moeilijker probleem is om de grenzen van de beschikbare gebieden te verkrijgen. De ondergrens is de set punten die nog steeds toegankelijk zijn, en de bovengrens is de set punten die niet toegankelijk zijn. In feite is dit eenvoudig: het is de grens van beschikbaarheid, gebaseerd op de randen van de boom van het kortste pad waarvoor het bronpunt van de rand toegankelijk is en het doelpunt van de rand is dat niet.

Hier is een voorbeeld

```
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))
```

CHAPTER 22

Python plug-ins voor QGIS server

- Server Filter Plugins architecture
  - requestReady
  - sendResponse
  - responseComplete
- Een uitzondering opwerpen vanuit een plug-in
- Een plug-in voor de server schrijven
  - Plug-inbestanden
  - __init__.py
  - HelloServer.py
  - De invoer aanpassen
  - De uitvoer aanpassen of vervangen
- Plug-in Access control
  - Plug-inbestanden
  - __init__.py
  - AccessControl.py
  - layerFilterExpression
  - layerFilterSubsetString
  - layerPermissions
  - authorizedLayerAttributes
  - allowToEdit
  - cacheKey

Python plugins can also run on QGIS Server (see *label_qgisserver*): by using the *server interface* (`QgsServerInterface`) a Python plugin running on the server can alter the behavior of existing core services (**WMS**, **WFS** etc.).

With the *server filter interface* (`QgsServerFilter`) we can change the input parameters, change the generated output or even by providing new services.

With the *access control interface* (`QgsAccessControlFilter`) we can apply some access restriction per requests.

22.1 Server Filter Plugins architecture

Server python plugins are loaded once when the FCGI application starts. They register one or more `QgsServerFilter` (from this point, you might find useful a quick look to the server plugins API docs). Each filter should implement at least one of three callbacks:

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

Alle filters hebben toegang tot het object voor het verzoek/antwoord (`QgsRequestHandler`) en kan al zijn eigenschappen bewerken (invoer/uitvoer) en exceptions opwerpen (hoewel op een bijzondere manier zoals we hieronder zullen zien).

Hier is een pseudocode die een typische serversessie weergeeft en wanneer de terugkoppelingen van het filter worden aangeroepen:

- **Haal het inkomende verzoek op**
  - maak afhandeling GET/POST/SOAP voor het verzoek
  - geef verzoek door aan een instantie van `QgsServerInterface`
  - roep plug-ins `requestReady()` filters aan
  - **indien er geen antwoord is**
    - **als SERVICE WMS/WFS/WCS is**
      - **maak WMS/WFS/WCS server**

        roep de servers `executeRequest()` aan en roep mogelijk aan `sendResponse()` plug-in filters bij stromende uitvoer of sla de byte stromende uitvoer en het type inhoud op in de afhandeling van het verzoek
    - roep plug-ins `responseComplete()` filters aan
  - roep plug-ins `sendResponse()` filters aan
  - afhandeling van het verzoek voert het antwoord uit

De volgende alinea's beschrijven de beschikbare terugkoppelingen tot in detail.

22.1.1 requestReady

Dit wordt aangeroepen als het verzoek gereed is: inkomende URL en gegevens zijn geparset en vóór te schakelen naar de bronservices (WMS, WFS etc.), is dit het punt waar u de invoer kunt bewerken en acties kunt uitvoeren als:

- authenticatie/autorisatie
- doorverwijzingen
- bepaalde parameters toevoegen/verwijderen (typenamen bijvoorbeeld)
- exceptions opwerpen

U zou zelfs een bronservice volledig kunnen vervangen door de parameter **SERVICE** te wijzigen en op die manier de bronservice volledig omzeilen (niet dat dat echter enige zin zou hebben).

22.1.2 sendResponse

Deze wordt aangeroepen wanneer de uitvoer wordt verzonden aan **FCGI** `stdout` (en van daaruit naar de cliënt), dit wordt normaal gesproken gedaan nadat bronservices hun proces hebben voltooid en nadat hook responseComplete werd aangeroepen, maar in een klein aantal gevallen kan de XML zo groot worden dat een stromende XML implementatie nodig was (WFS GetFeature is één ervan), in dit geval werd `sendResponse()` meerdere keren aangeroepen voordat het antwoord volledig was (en vóórdat `responseComplete()` werd aangeroepen). De voor de hand liggende consequentie is dat `sendResponse()` normala gesproken eenmaal wordt aangeroepen maar zou bij uitzondering meerdere keren aangeroepen kunnen worden en in dat geval (en alleen in dat geval) wordt het ook aangeroepen vóór `responseComplete()`.

`sendResponse()` is de beste plaats voor het direct bewerken van de uitvoer van bronservices en hoewel `responseComplete()` gewoonlijk ook een optie is, is `sendResponse()` de enige geldige optie in het geval van stromende services.

22.1.3 responseComplete

Dit wordt eenmaal aangeroepen wanneer de bronservices (indien aangesproken) hun proces voltooien en het verzoek gereed is om te worden verzonden naar de cliënt. Zoals hierboven besproken wordt dit normaal gesproken aangeroepen vóór `sendResponse()` met uitzondering van stromende services (of andere filters voor plug-ins) die `sendResponse()` eerder zouden hebben kunnen aangeroepen.

`responseComplete()` is de ideale plek om implementatie voor nieuwe services te verschaffen (WPS of aangepaste services) en om de uitvoer, komende vanaf bronservices, direct te bewerken (bijvoorbeeld om ene watermerk aan een afbeelding van WMS toe te voegen).

22.2 Een uitzondering opwerpen vanuit een plug-in

Enig werk moet voor dit onderwerp nog worden gedaan: de huidige implementatie kan onderscheid maken tussen afgehandelde en niet afgehandelde uitzonderingen door het instellen van een eigenschap `QgsRequestHandler` voor een instantie van `QgsMapServiceException`, op deze manier kan de hoofdcode van C++ de afgehandelde uitzonderingen van Python afvangen en niet afgehandelde uitzonderingen negeren (of beter nog: ze loggen).

Deze benadering werkt in de basis maar is nog niet erg "Pythonisch": een betere benadering zou zijn om uitzonderingen op te werpen vanuit de code van Python en ze op zien borrelen in een lus van C++ om daar te worden afgehandeld.

22.3 Een plug-in voor de server schrijven

A server plugins is just a standard QGIS Python plugin as described in *Python plug-ins ontwikkelen*, that just provides an additional (or alternative) interface: a typical QGIS desktop plugin has access to QGIS application through the `QgisInterface` instance, a server plugin has also access to a `QgsServerInterface`.

Een speciaal item voor metadata is nodig (in *metadata.txt*) om QGIS Server te vertellen dat een plug-in een interface voor de server heeft:

```
server=True
```

The example plugin discussed here (with many more example filters) is available on github: QGIS HelloServer Example Plugin

22.3.1 Plug-inbestanden

Hier is de mappenstructuur van onze voorbeeld-plug-in voor de server

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

22.3.2 __init__.py

This file is required by Python's import system. Also, QGIS Server requires that this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin *__init__.py* looks like:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from HelloServer import HelloServerServer
    return HelloServerServer(serverIface)
```

22.3.3 HelloServer.py

Dit is waar de magie gebeurt en dit is hoe de magie eruit ziet: (bijv. `HelloServer.py`)

Een plug-in voor de server bestaat gewoonlijk uit één of meer callbacks, verpakt in objecten, genaamd QgsServer-Filter.

Elk `QgsServerFilter` implementeert één of meer van de volgende callbacks:

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

Het volgende voorbeeld implementeert een minimaal filter dat *HelloServer!* afdrukt in het geval dat de parameter **SERVICE** gelijk is aan "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!')
```

De filters moeten worden geregistreerd in de **serverIface** zoals in het volgende voorbeeld:

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

De tweede parameter van `registerFilter()` maakt het mogelijk een prioriteit in te stellen die de volgorde definieert voor de callbacks met dezelfde naam (de laagste prioriteit wordt het eerst uitgevoerd).

Door de drie callbacks te gebruiken, kunnen plug-ins de invoer en/of de uitvoer van de server op veel verschillende manieren manipuleren. Op elk moment heeft de instantie van de plug-in toegang tot de `QgsRequestHandler` via de `QgsServerInterface`, de `QgsRequestHandler` heeft veel methoden die kunnen worden gebruikt om de parameters voor de invoer te wijzigen vóór de bronverwerking door de server (door `requestReady()` te gebruiken) of nadat het verzoek is verwerkt door de bronservices (door `sendResponse()` te gebruiken).

De volgende voorbeelden behandelen enkele veel voorkomende gevallen van gebruik:

22.3.4 De invoer aanpassen

De voorbeeld plug-in bevat een testvoorbeeld dat parameters voor invoer wijzigt die afkomstig zijn uit de tekenreeks van de query, in dit voorbeeld wordt een nieuwe parameter ingevoerd in de (reeds geparste) `parameterMap`, deze parameter is dan zichtbaar voor bronservices (WMS etc.), aan het einde van de verwerking door bronservices controleren we of de parameter er nog steeds is:

```
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
```

Dit is een extract van wat u ziet in het logbestand:

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServ
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 p
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&re
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms] inserting pair
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter plugin default reque
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.req
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default configuration file path: 
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking byte array is ok t
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, sett
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.res
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - Param
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFi
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.sen
```

Op de geaccentueerde regel geeft de tekenreeks "SUCCESS" aan dat de plug-in voor de test is geslaagd.

Dezelfde techniek kan worden gebruikt om een aangepaste service te gebruiken in plaats van een bronservice: u zou bijvoorbeeld een verzoek **WFS SERVICE** kunnen overslaan of elk ander bronverzoek door slechts de parameter **SERVICE** naar iets anders te wijzigen en de bronservice zal worden overgeslagen, dan kunt u uw aangepaste resultaten invoeren in de uitvoer en die naar de cliënt verzenden (dat is hieronder uitgelegd).

22.3.5 De uitvoer aanpassen of vervangen

Het voorbeeld watermark filter laat zien hoe de uitvoer van WMS te vervangen door een nieuwe afbeelding die wordt verkregen door het toevoegen van een afbeelding van een watermerk bovenop de afbeelding van WMS die werd gegenereerd door de bronservice van 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)
```

In dit voorbeeld is de waarde van de parameter **SERVICE** gecontroleerd en als het inkomende verzoek een **WMS GETMAP** is en er geen uitzonderingen zijn ingesteld door een eerder uitgevoerde plug-in of door de bronservice (WMS in dit geval), wordt de door WMS gegenereerde afbeelding opgehaald uit de buffer voor de uitvoer en wordt de afbeelding van het watermerk toegevoegd. De laatste stap is om de buffer voor de uitvoer op te schonen en die te vervangen door de nieuw gegenereerde afbeelding. Onthoud dat in een situatie in de echte wereld we ook het type van de verzochte afbeelding zouden controleren in plaats van PNG in elk geval terug te geven.

22.4 Plug-in Access control

22.4.1 Plug-inbestanden

Hier is de mappenstructuur van onze voorbeeld-plug-in voor de server:

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

22.4.2 __init__.py

This file is required by Python's import system. As for all QGIS server plugins, this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin *__init__.py* looks like:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from MyAccessControl.AccessControl import AccessControl
    return AccessControl(serverIface)
```

22.4.3 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()
```

Dit voorbeeld geeft een voorbeeld voor volledige toegang voor iedereen.

Het is de rol van de plug-in om te weten wie er is ingelogd.

Voor al deze methoden hebben de laag als argument om in staat te zien om de rechten per laag aan te passen.

22.4.4 layerFilterExpression

Gebruikt om een Expressie toe te voegen om de resultaten te beperken, bijv.:

```
def layerFilterExpression(self, layer):
    return "$role = 'user'"
```

Te beperken tot de mogelijkheid waar de rol attribuut gelijk is aan "user".

22.4.5 layerFilterSubsetString

Hetzelfde als hiervoor maar dan door de `SubsetString` te gebruiken (uitgevoerd in de database)

```
def layerFilterSubsetString(self, layer):
    return "role = 'user'"
```

Te beperken tot de mogelijkheid waar de rol attribuut gelijk is aan "user".

22.4.6 layerPermissions

Toegang beperken tot de laag.

Geef een object terug van het type `QgsAccessControlFilter.LayerPermissions`, die de eigenschappen heeft:

- `canRead` om hem te zien in de `GetCapabilities` en rechten voor lezen hebben.
- `canInsert` om een nieuw object in te kunnen voeren.
- `canUpdate` om een object bij te kunnen werken.
- `candelete` om een object te kunnen verwijderen.

Voorbeeld:

```
def layerPermissions(self, layer):
    rights = QgsAccessControlFilter.LayerPermissions()
    rights.canRead = True
    rights.canRead = rights.canInsert = rights.canUpdate = rights.canDelete = False
    return rights
```

Om alles te beperken tot toegang voor alleen-lezen.

22.4.7 authorizedLayerAttributes

Gebruikt om de zichtbaarheid van een specifieke subset van attributen te beperken.

Het argument attribute geeft de huidige set van zichtbare attributen terug.

Voorbeeld:

```
def authorizedLayerAttributes(self, layer, attributes):
    return [a for a in attributes if a != "role"]
```

Het attribuut 'role' verbergen.

22.4.8 allowToEdit

Dit wordt gebruikt om het bewerken van een subset van objecten te beperken.

Het wordt gebruikt in het protocol `WFS-Transaction`.

Voorbeeld:

```
def allowToEdit(self, layer, feature):
    return feature.attribute('role') == 'user'
```

Om het mogelijk te maken alleen objecten te bewerken die het attribuut role hebben met de waarde user.

22.4.9 cacheKey

QGIS server onderhoudt een cache van de capabilities, om dan een cache per rol te hebben kunt u de rol teruggeven met deze methode. Of geef `None` terug om de cache volledig uit te schakelen.

Index

API, 1
Authenticatie DB, **72**
Authenticatie-database, **72**
Authenticatiemethode, **72**

Calculating values, 50
Categorized symbology renderer, 27
Coördinaten ReferentieSystemen, 39
Configuratie voor authenticatie, **72**
Console
    Python, 2
Custom
    Renderer, 31
Custom applications
    Python, 3
    Running, 4

Delimited text files
    Loading, 10

Environment
    PYQGIS_STARTUP, 1
Expressions, 50
    Evaluating, 52
    Parsing, 52

Filtering, 50

Geometry
    Access to, 36
    Construction, 35
    Handling, 33
    Predicates and operations, 36
GPX files
    Loading, 10
Graduated symbol renderer, 28

Hoofdwachtwoord, **72**

Iterating features, 18

Loading
    Delimited text files, 10
    GPX files, 10
    MySQL geometries, 10
    OGR layers, 9
    PostGIS layers, 9
    Projects, 7
    Raster layers, 11
    SpatiaLite layers, 10
    Vector layers, 9
    WFS vector, 10
    WMS raster, 11

Map canvas, 40
    Custom canvas items, 45
    Custom map tools, 44
    Embedding, 41
    Map tools, 42
    Rubber bands, 43
    Vertex markers, 43
map canvas
    architecture, 41
Map layer registry, 11
    Adding a layer, 11
Map printing, 46
Map rendering, 46
    Simple, 47
Memory layer, 24
Metadata, 107
metadata, 107
metadata.txt, 63, 107
MySQL geometries
    Loading, 10

Objecten selecteren, 17
OGR layers
    Loading, 9
Output
    PDF, 50
    Raster image, 50
    Using Map Composer, 48

Plugin layers, 84
    Sub-klassen in QgsPluginLayer, 85
Plugins
    Access attributes of selected features, 93
    Adding shortcut, 93
    Code snippets, 67
    Debugging, 78
    Developing, 59, 69

Documentatie, 66
Implementing help, 66
Initialisation, 64
Metadata, 63
metadata.txt, 107
Officiële Python plug-in opslagplaats, 90
Processing algorithm, 94
Releasing, 87
Resource file, 66
Toggle layers, 93
User interaction, 56
Vertaling, 67
Writing, 62
Writing code, 62

plugins
testing, 84

PostGIS layers
Loading, 9

Projections, 40

Projects
Loading, 7

PyQGIS
Vector layers, 17

PYQGIS_STARTUP
Environment, 1

Python
Console, 2
Custom applications, 3
Developing plugins, 59
Developing server plugins, 104
Infrastructuur voor authenticatie, 69
Plugins, 2
Standalone scripts, 3
startup, 1
startup.py, 2

Querying
Raster layers, 15

Raster
Raster layers, 12

Raster layers
Details, 13
Loading, 11
Multi band, 14
Querying, 15
Raster, 12
Refreshing, 15
Renderer, 13
Single band, 14

Refreshing
Raster layers, 15

Renderer
Custom, 31

resources.qrc, 66

Running
Custom applications, 4

Server plugins
Developing, 104

server plugins
metadata.txt, 107

Settings
Global, 55
Map layer, 56
Project, 55
Reading, 53
Storing, 53

Single symbol renderer, 26

Spatial index, 23

SpatiaLite layers
Loading, 10

Standalone scripts
Python, 3

startup
Python, 1

Symbol layers
Creating custom types, 29
Working with, 29

Symbology
Categorized symbol renderer, 27
Graduated symbol renderer, 28
Single symbol renderer, 26

Symbols
Working with, 28

Vector layers
Creating, 23
Editing, 20
Loading, 9
Symbology, 25

WFS vector
Loading, 10

WMS raster
Loading, 11