PyQGIS developer cookbook

Release 3.4

QGIS Project

mrt. 15, 2020

Contents

1 Introductie **1**
1.1 Scripten in de console voor Python . . . 2
1.2 Plug-ins in Python . . . 2
1.3 Python-code uitvoeren wanneer QGIS start . . . 3
1.4 Toepassingen in Python . . . 3
1.5 Technische opmerkingen over PyQt en SIP . . . 6

2 Projecten laden **7**

3 Lagen laden **9**
3.1 Vectorlagen . . . 9
3.2 Rasterlagen . . . 12
3.3 Instance QgsProject . . . 13

4 Rasterlagen gebruiken **15**
4.1 Details laag . . . 15
4.2 Renderer . . . 16
4.3 Waarden bevragen . . . 17

5 Vectorlagen gebruiken **19**
5.1 Informatie over attributen ophalen . . . 20
5.2 Itereren over vectorlagen . . . 20
5.3 Objecten selecteren . . . 21
5.4 Vectorlagen bewerken . . . 23
5.5 Ruimtelijke index gebruiken . . . 26
5.6 Vectorlagen maken . . . 27
5.7 Uiterlijk (symbologie) van vectorlagen . . . 30
5.8 Meer onderwerpen . . . 39

6 Afhandeling van geometrie **41**
6.1 Construeren van geometrie . . . 41
6.2 Toegang tot geometrie . . . 42
6.3 Predicaten en bewerking voor geometrieën . . . 43

7 Ondersteuning van projecties **45**
7.1 Coördinaten ReferentieSystemen . . . 45
7.2 CRS transformatie . . . 46

8 Het kaartvenster gebruiken **49**
8.1 Kaartvenster inbedden . . . 50
8.2 Elastieken banden en markeringen voor punten . . . 51
8.3 Gereedschappen voor de kaart gebruiken in het kaartvenster . . . 52

8.4 Aangepaste gereedschappen voor de kaart schrijven . . . 53
8.5 Aangepaste items voor het kaartvenster schrijven . . . 54

9 Kaart renderen en afdrukken **55**
9.1 Eenvoudig renderen . . . 55
9.2 Lagen met een verschillend CRS renderen . . . 56
9.3 Uitvoer door Afdruklay-out te gebruiken . . . 56

10 Expressies, filteren en waarden berekenen **59**
10.1 Parsen van expressies . . . 60
10.2 Evalueren van expressies . . . 60
10.3 Fouten in expressies afhandelen . . . 62

11 Instellingen lezen en opslaan **63**

12 Communiceren met de gebruiker **65**
12.1 Berichten weergeven. De klasse QgsMessageBar . . . 65
12.2 Voortgang weergeven . . . 67
12.3 Loggen . . . 68

13 Infrastructuur voor authenticatie **71**
13.1 Introductie . . . 72
13.2 Woordenlijst . . . 72
13.3 QgsAuthManager is het toegangspunt . . . 72
13.4 Plug-ins aanpassen om de infrastructuur voor authenticatie te gebruiken . . . 75
13.5 GUI's voor authenticatie . . . 76

14 Taken - veel werk op de achtergrond doen **79**
14.1 Introductie . . . 79
14.2 Voorbeelden . . . 80

15 Python plug-ins ontwikkelen **85**
15.1 Plug-ins voor Python structureren . . . 85
15.2 Codesnippers . . . 94
15.3 Plug-in-lagen gebruiken . . . 95
15.4 Instellingen voor de IDE voor het schrijven en debuggen van plug-ins . . . 96
15.5 Uw plug-in uitgeven . . . 101

16 Een plug-in voor Processing schrijven **105**

17 Bibliotheek Netwerkanalyse **107**
17.1 Algemene informatie . . . 107
17.2 Een grafiek bouwen . . . 108
17.3 Grafiekanalyse . . . 110

18 Python plug-ins voor QGIS server **115**
18.1 Server Filter Plugins architecture . . . 116
18.2 Een uitzondering opwerpen vanuit een plug-in . . . 117
18.3 Een plug-in voor de server schrijven . . . 117
18.4 Plug-in Access control . . . 121

19 Cheatsheet voor PyQGIS **125**
19.1 Gebruikersinterface . . . 125
19.2 Instellingen . . . 125
19.3 Werkbalken . . . 125
19.4 Menu's . . . 126
19.5 Kaartvenster . . . 126
19.6 Lagen . . . 126
19.7 Inhoud . . . 130
19.8 Uitgebreide inhoud . . . 130

19.9 Algoritmes voor Processing 132
19.10 Decoraties 133
19.11 Bronnen 134

CHAPTER 1

Introductie

Dit document is zowel bedoeld om te gebruiken als handleiding en als gids met verwijzingen. Hoewel het niet alle mogelijke gevallen van gebruik weergeeft zou het een goed overzicht moeten geven van de belangrijkste functionaliteiten.

- *Scripten in de console voor Python*
- *Plug-ins in Python*
- *Python-code uitvoeren wanneer QGIS start*
  - *Het bestand `startup.py`*
  - *De omgevingsvariabele PYQGIS_STARTUP*
- *Toepassingen in Python*
  - *PyQGIS gebruiken in zelfstandige scripts*
  - *PyQGIS gebruiken in aangepaste toepassing*
  - *Aangepaste toepassingen uitvoeren*
- *Technische opmerkingen over PyQt en SIP*

Ondersteuning voor Python werd voor het eerst geïntroduceerd in QGIS 0.9. Er zijn verscheidene manieren om Python te gebruiken QGIS Desktop (worden in de volgende gedeelten behandeld):

- Opdrachten opgeven in de console voor Python in QGIS
- Plug-ins in Python maken en gebruiken
- Python-code automatisch uitvoeren wanneer QGIS start
- Aangepaste toepassingen maken, gebaseerd op de API van QGIS

Python-bindings zijn ook beschikbaar voor QGIS Server, inclusief plug-ins voor Python (zie *Python plug-ins voor QGIS server*) en Python-bindings die kunnen worden gebruikt om QGIS Server in te bedden in een toepassing van Python.

Er is een verwijzing complete API voor QGIS dat de klassen uit de bibliotheken van QGIS documenteert. The Pythonic QGIS API (pyqgis) is nagenoeg identiek aan de API voor C++.

Een goede bron voor het leren hoe algemene taken uit te voeren is om bestaande plug-ins te downloaden vanaf de opslagplaats voor plugins en de code ervan te bestuderen.

1.1 Scripten in de console voor Python

QGIS verschaft een geïntegreerde Python console voor scripten. Deze kan geopend worden via het menu *Plug-ins* → *Python Console*:

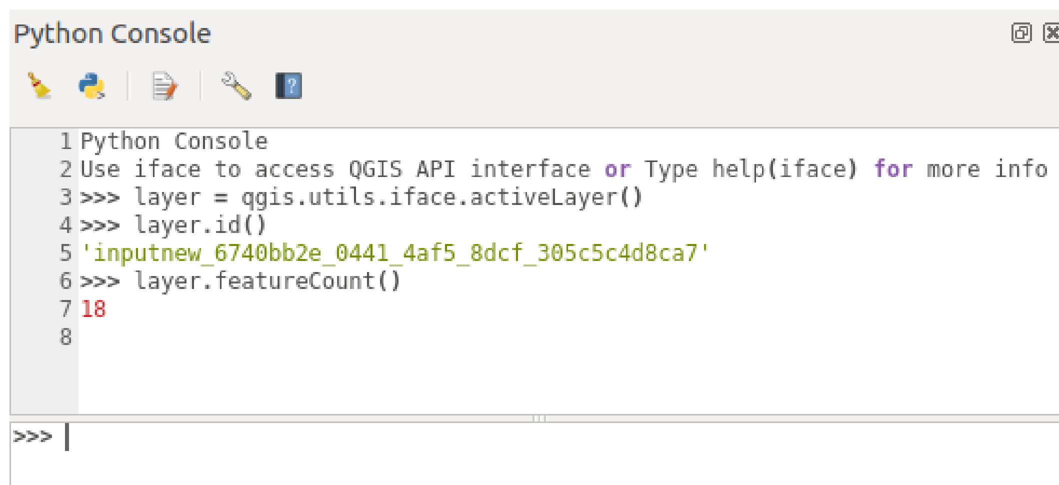


Figure 1.1: QGIS Python-console

De schermafdruk hierboven illustreert hoe de huidige geselecteerde laag in de Lagenlijst te verkrijgen, de ID ervan weer te geven en optioneel, als het een vectorlaag is, het aantal objecten weer te geven. Voor interactie met de omgeving van QGIS is er een variabele `iface`, wat een instance is van `QgsInterface`. Deze interface maakt toegang mogelijk tot het kaartvenster, menu's, werkbalken en andere delen van de toepassing QGIS.

Voor het gemak van de gebruiker zullen de volgende argumenten worden uitgevoerd wanneer de console wordt opgestart (in de toekomst zal het mogelijk zijn meer initiële opdrachten in te stellen)

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

Voor hen die de console vaak gebruiken, kan het handig zijn een sneltoets in te stellen voor het activeren van de console (in *Extra* → *Toetsenbord sneltoetsen…*)

1.2 Plug-ins in Python

De functionaliteit van QGIS kan worden uitgebreid met plug-ins. Plug-ins mogen zijn geschreven in Python. Het belangrijkste voordeel boven plug-ins van C++ is de eenvoudige manier van verdelen (niet meer nodig om te compileren voor elk platform) en eenvoudiger ontwikkelen.

Veel plug-ins, die verschillende functionaliteiten behandelen, zijn geschreven sinds de introductie van ondersteuning voor Python. Het installatieprogramma voor plug-ins stelt gebruikers in staat om eenvoudig plug-ins voor Python op te halen, bij te werken en te verwijderen. Bekijk de pagina Python Plugins voor meer informatie over plug-ins en het ontwikkelen van plug-ins.

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

Notitie: Plug-ins voor Python zijn ook beschikbaar voor QGIS Server. Bekijk *Python plug-ins voor QGIS server* voor meer details.

1.3 Python-code uitvoeren wanneer QGIS start

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

1. Een script startup.py schrijven
2. Instellen van de omgevingsvariabele `PYQGIS_STARTUP` naar een bestaand bestand voor Python

1.3.1 Het bestand `startup.py`

Elke keer als QGIS start, wordt in de thuismap voor Python van de gebruiker

- Linux: `.local/share/QGIS/QGIS3`
- Windows: `AppData\Roaming\QGIS\QGIS3`
- macOS: `Library/Application Support/QGIS/QGIS3`

gezocht naar een bestand genaamd `startup.py`. Als dat bestand bestaat wordt het door de ingebedde interpreter van Python uitgevoerd.

Notitie: Het standaard pad is afhankelijk van het besturingssysteem. Open, om het pad te zoeken dat voor u zal werken, de console voor Python en voer `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` uit om de lijst met de standaard mappen te zien.

1.3.2 De omgevingsvariabele PYQGIS_STARTUP

U kunt Python-code uitvoeren kort voor de initialisatie van QGIS wordt voltooid door de omgevingsvariabele `PYQGIS_STARTUP` in te stellen op het pad van een bestaand bestand van Python.

Deze code zal worden uitgevoerd vóórdat de initialisatie van QGIS is voltooid. Deze methode is zeer handig voor het opschonen van sys.path, wat ongewenste paden zou kunnen bevatten, of voor het isoleren/laden van de initiële omgeving zonder een virtuele omgeving te vereisen, bijv. homebrew of installaties van MacPorts op Mac.

1.4 Toepassingen in Python

Het is vaak handig om enkele scripts te maken voor het automatiseren van processen. Met PyQGIS is dit perfect mogelijk — importeer de module `qgis.core`, initialiseer die en u bent klaar om te verwerken.

Of u wilt misschien een interactieve toepassing maken die functionaliteit van GIS gebruikt — metingen uitvoeren, exporteren van een kaart als PDF, …. De module `qgis.gui` geeft verscheidene componenten voor een GUI, waarvan de meest belangrijke de widget voor het kaartvenster is die kan worden opgenomen in de toepassing met ondersteuning voor zoomen, pannen en/of elke andere aangepaste gereedschappen voor de kaart.

Aangepaste toepassingen of zelfstandige scripts voor PyQGIS moeten worden geconfigureerd om de bronnen van QGIS te kunnen vinden, zoals informatie over de projectie en providers voor het lezen van vector- en rasterlagen. Bronnen voor QGIS worden geïnitialiseerd door een aantal regels toe te voegen aan het begin van uw toepassing of script. De code om QGIS voor aangepaste toepassingen en zelfstandige scripts te initialiseren is soortgelijk. Voorbeelden voor elk daarvan worden hieronder vermeld.

Notitie: Gebruik *niet* `qgis.py` als naam voor uw script. Python zal niet in staat zijn de bindingen te importeren omdat de naam van het script die zal overschaduwen.

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:

```
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.

# Finally, exitQgis() is called to remove the
# provider and layer registries from memory

qgs.exitQgis()
```

Eerst importeren we de module `qgis.core` en configureren dan het pad voor het voorvoegsel. 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` wordt ingesteld op `True` en specificeert dat 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 *Scripten in de console voor Python* 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 specificeert 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 `True` op in plaats van `False` om aan te geven dat we van plan zijn om een GUI te gaan gebruiken.

```
from qgis.core import *

# Supply the path to the 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.  We need
# this 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.

# Finally, exitQgis() is called to remove the
# provider and layer registries from memory
qgs.exitQgis()
```

Nu kunt u werken met de API van QGIS - lagen laden en enige verwerking doen of een GUI met een kaartvenster opstarten. De mogelijkheden zijn eindeloos :-)

1.4.3 Aangepaste toepassingen uitvoeren

U moet uw systeem vertellen waar te zoeken naar de bibliotheken van QGIS en de toepasselijke modules voor Python als zij nog niet op een bekende locatie staan - anders zal Python gaan klagen:

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

Dit kan worden opgelost door de omgevingsvariabele `PYTHONPATH` in te stellen. In de volgende opdrachten zou `<qgispath>` moeten worden vervangen door uw actuele pad voor de installatie van QGIS:

- op Linux: **`export PYTHONPATH=/<qgispath>/share/qgis/python`**
- op Windows: **`set PYTHONPATH=c:\<qgispath>\python`**
- op macOS: **`export PYTHONPATH=/<qgispath>/Contents/Resources/python`**

Nu is het pad naar de modules van PyQGIS bekend, maar zij zijn afhankelijk van de bibliotheken `qgis_core` en `qgis_gui` (de modules van Python dienen slechts als verpakkingen). Het pad naar deze bibliotheken zou onbekend kunnen zijn voor het besturingssysteem, en dan zult u opnieuw een fout bij het importeren krijgen (het bericht kan variëren, afhankelijk van het systeem):

```
>>> import qgis.core
ImportError: libqgis_core.so.3.2.0: 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:

- op Linux: **`export LD_LIBRARY_PATH=/<qgispath>/lib`**
- op Windows: **`set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%`** waar `<qgisrelease>` zou moeten worden vervangen door het type uitgave dat uw doel is (bijv, `qgis-ltr`, `qgis`, `qgis-dev`)

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:

- eis dat de gebruiker QGIS installeert, voorafgaand aan het installeren van uw toepassing. Het installatieprogramma van de toepassing zou moeten zoeken naar standaardlocaties voor de bibliotheken van QGIS en de gebruiker moeten toestaan het pad in te vullen als dat niet werd gevonden. Deze benadering heeft het voordeel dat het eenvoudiger is, het vereist echter dat de gebruiker meer stappen uitvoert.
- 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. U kunt zelfstandige toepassingen uitrollen op Windows en MacOS, maar voor Linux de installatie van GIS overlaten aan de gebruiker en diens pakketbeheer.

1.5 Technische opmerkingen over PyQt en SIP

We hebben gekozen voor Python omdat het één van de meest favoriete talen voor scripten is. Bindingen voor PyQGIS in QGIS 3 zijn afhankelijk van SIP en PyQt5. De reden voor het gebruiken van SIP in plaats van het meer breder gebruikte SWIG is dat de gehele code voor QGIS afhankelijk is van bibliotheken van Qt. Bindingen voor Python voor Qt (PyQt) worden gedaan met SIP en dat maakt een naadloze integratie van PyQGIS met PyQt mogelijk.

CHAPTER 2

Projecten laden

Soms moet u een bestaand project uit een plug-in laden of (nog vaker) bij het ontwikkelen van een zelfstandige toepassing in Python voor QGIS (zie: *Toepassingen in Python*).

U dient een instance te maken van de klasse `QgsProject` om een project in de huiidge toepassing QGIS te laden . Dit is een klasse singleton, dus u moet eerst de methode `instance()` ervan gebruiken om dat te doen. U kunt de methode `read()` ervan aanroepen, waarin het pad van het te laden project wordt doorgegeven:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt classes you will use in this script as shown below:
from qgis.core import QgsProject
# 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())
'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read('/home/user/projects/my_other_qgis_project.qgs')
print(project.fileName())
'/home/user/projects/my_other_qgis_project.qgs'
```

Als u aanpassingen moet maken aan het project (bijvoorbeeld enige lagen toevoegen of verwijderen) en uw wijzigingen opslaan, roep de methode `write()` van uw instance voor het project aan. De methode `write()` accepteert ook een optioneel pad voor het opslaan van het project op een nieuwe locatie:

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

Beide functies `read()` en `write()` geven een Booleaanse waarde terug die u kunt gebruiken om te controleren of de bewerking succesvol was.

Notitie: U dient, als u een zelfstandige toepassing voor QGIS schrijft, een klasse `QgsLayerTreeMapCanvasBridge` te instantiëren zoals in het voorbeeld hieronder om het geladen project te synchroniseren met het kaartvenster:

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

CHAPTER 3

Lagen laden

De codesnippers op deze pagina hebben de volgende import nodig:

```
import os # This is is needed in the pyqgis console also
from qgis.core import (
    QgsVectorLayer
)
```


- *Vectorlagen*
- *Rasterlagen*
- *Instance QgsProject*

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

3.1 Vectorlagen

Specificeer de identificatie van de gegevensbron van de laag, de naam voor de laag en de naam van de provider om een instance voor een vectorlaag te maken:

```
# get the path to the shapefile e.g. /home/project/data/ports.shp
path_to_ports_layer = os.path.join(QgsProject.instance().homePath(), "data", "ports
↪", "ports.shp")

# The format is:
# vlayer = QgsVectorLayer(data_source, layer_name, provider_name)

vlayer = QgsVectorLayer(path_to_ports_layer, "Ports layer", "ogr")
if not vlayer.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.

Voor een laag van GeoPackage:

```
# get the path to a geopackage  e.g. /home/project/data/data.gpkg
path_to_gpkg = os.path.join(QgsProject.instance().homePath(), "data", "data.gpkg")
# append the layername part
gpkg_places_layer = path_to_gpkg + "|layername=places"
# e.g. gpkg_places_layer = "/home/project/data/data.gpkg|layername=places"
vlayer = QgsVectorLayer(gpkg_places_layer, "Places layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")
```

De snelste manier om een vectorlaag te openen en weer te geven in QGIS is de `addVectorLayer()` van `QgisInterface`:

```
vlayer = iface.addVectorLayer(path_to_ports_layer, "Ports layer", "ogr")
if not vlayer:
  print("Layer failed to load!")
```

Dit maakt een nieuwe laag en voegt die toe aan het huidige project van QGIS (waardoor het verschijnt in de lagenlijst). De functie geeft de instance van de laag terug of `None` als de laag niet kon worden geladen.

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

- bibliotheek OGR (Shapefiles en vele andere bestandsindelingen) — gegevensbron is het pad naar het bestand:
  - vpor Shapefile:

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

  - voor DXF (let op de interne opties in de URI van de gegevensbron):

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

- database PostGIS - gegevensbron is een tekenreeks met alle benodigde informatie om een verbinding naar een database van PostgreSQL te maken.

  De klasse `QgsDataSourceUri` kan deze string voor u maken. Onthoud dat QGIS moet worden gecompileerd met ondersteuning voor Postgres, anders is deze provider niet beschikbaar:

  ```
  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 of andere gescheiden tekstbestanden — om een bestand te openen met een punt-komma als scheidingsteken, met veld "x" voor de X-coördinaat en veld "y" voor de Y-coördinaat zou u zoiets als dit gebruiken:

  ```
  uri = "/some/path/file.csv?delimiter={}&xField={}&yField={}".format(";", "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={}&crs=epsg:4723&wktField={}".
↪format(";", "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")
```

- database SpatiaLite — Soortgelijk aan databases van PostGIS, `QgsDataSourceUri` kan worden gebruikt voor het maken van de identificatie van de gegevensbron:

```
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&request=GetFeature&service=WFS",
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: U kunt de gegevensbron van een bestaande laag wijzigen door `setDataSource()` op een instance van `QgsVectorLayer` aan te roepen, zoals in het volgende voorbeeld:

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

3.2 Rasterlagen

Voor toegang tot rasterbestanden wordt de bibliotheek GDAL gebruikt. Het ondersteunt een breed scala aan bestandsindelingen. In het geval u problemen hebt met het openen van enkele bestanden, controleer dan of uw GDAL ondersteuning heeft voor die bepaalde indeling (niet alle indelingen zijn standaard beschikbaar). Specificeer zijn bestandsnaam en weergavenaam om een raster uit een bestand te laden:

```
# get the path to a tif file  e.g. /home/project/data/srtm.tif
path_to_tif = os.path.join(QgsProject.instance().homePath(), "data", "srtm.tif")
rlayer = QgsRasterLayer(path_to_tif, "SRTM layer name")
if not rlayer.isValid():
    print("Layer failed to load!")
```

Raster laden vanuit een GeoPackage

```
# get the path to a geopackage  e.g. /home/project/data/data.gpkg
path_to_gpkg = os.path.join(QgsProject.instance().homePath(), "data", "data.gpkg")
# gpkg_raster_layer = "GPKG:/home/project/data/data.gpkg:srtm"
gpkg_raster_layer = "GPKG:" + path_to_gpkg + ":srtm"

rlayer = QgsRasterLayer(gpkg_raster_layer, "layer name you like", "gdal")

if not rlayer.isValid():
    print("Layer failed to load!")
```

Soortgelijk aan vectorlagen kunnen rasterlagen worden geladen met de functie addRasterLayer van het object `QgisInterface`:

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

Dit maakt een nieuwe laag en voegt die in één stap toe aan het huidige project (waardoor het verschijnt in de lagenlijst).

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')
```

Hier is een beschrijving van de parameters die de URI voor WCS mag bevatten:

De URI voor WCS is samengesteld uit paren **sleutel=waarde**, gescheiden door `&`. Het is dezelfde indeling als een tekenreeks voor een query in een URL, op dezelfde manier gecodeerd. `QgsDataSourceUri` zou moeten worden gebruikt om de URI te construeren om er voor te zorgen dat speciale tekens op de juiste manier worden gecodeerd.

- **url** (vereist) : WCS Server URL. Gebruik geen VERSION in URL, omdat elke versie van WCS een andere naam voor de parameter voor de versie **GetCapabilities** gebruikt, zie param versie.
- **identifier** (vereist) : Bedekkingsnaam
- **time** (optioneel) : tijdspositie of tijdsperiode (beginPosition/endPosition[/timeResolution])
- **format** (optioneel) : Ondersteunde naam voor indeling. Standaard is de eerste ondersteunde indeling met tif in de naam of de eerste ondersteunde indeling.
- **crs** (optioneel) : CRS in de vorm AUTHORITY:ID, bijv. EPSG:4326. Standaard is EPSG:4326 indien ondersteund of het eerste ondersteunde CRS.
- **username** (optioneel) : Gebruikersnaam voor basisauthenticatie.
- **password** (optioneel) : Wachtwoord voor basisauthenticatie.

- **IgnoreGetMapUrl** (optioneel, hack) : Indien gespecificeerd (ingesteld op 1), negeer GetCoverage URL aangeboden door GetCapabilities. Kan nodig zijn als een server niet juist is geconfigureerd.
- **InvertAxisOrientation** (optioneel, hack) : Indien gespecificeerd (ingesteld op 1), schakel de as in het verzoek GetCoverage. Kan nodig zijn voor geografisch CRS als een server de verkeerde volgorde voor assen gebruikt.
- **IgnoreAxisOrientation** (optioneel, hack) : Indien gespecificeerd (ingesteld op 1), as-oriëntatie niet omdraaien overeenkomstig de standaard van WCS voor geografisch CRS.
- **cache** (optioneel) : cache laadbeheer, zoals beschreven in QNetworkRequest::CacheLoadControl, maar verzoek wordt opnieuw verzonden als PreferCache indien mislukt met AlwaysCache. Toegestane waarden: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. Standaard is AlwaysCache.

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:4326'
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print("Layer failed to load!")
```

3.3 Instance QgsProject

Als u de geopende lagen wilt gebruiken voor renderen, vergeet dan niet om ze toe te voegen aan de instance `QgsProject`. De instance `QgsProject` wordt eigenaar van de lagen en er kan later toegang toe worden verkregen vanuit elk deel van de toepassing door hun unieke ID. Als de laag wordt verwijderd uit het project, wordt hij ook verwijderd. Lagen kunnen door de gebruiker worden verwijderd in de interface van QGIS interface, of via Python met de methode `removeMapLayer()`.

Toevoegen van een laag aan het huidige project wordt gedaan met de methode `addMapLayer()`:

```
QgsProject.instance().addMapLayer(rlayer)
```

Een laag op een absolute positie toevoegen:

```
# first add the layer without showing it
QgsProject.instance().addMapLayer(rlayer, False)
# obtain the layer tree of the top-level group in the project
layerTree = iface.layerTreeCanvasBridge().rootGroup()
# the position is a number starting from 0, with -1 an alias for the end
layerTree.insertChildNode(-1, QgsLayerTreeLayer(rlayer))
```

Als u de laag wilt verwijderen, gebruik dan de methode `removeMapLayer()`:

```
# QgsProject.instance().removeMapLayer(layer_id)
QgsProject.instance().removeMapLayer(rlayer.id())
```

In de bovenstaande code wordt de laag-ID doorgegeven (die kunt u zien door het aanroepen van de methode `id()` van de laag), maar u kunt ook het object laag zelf doorgeven.

Voor een lijst met geladen lagen en laag-ID's, gebruik de methode `mapLayers()`:

```
QgsProject.instance().mapLayers()
```

CHAPTER 4

Rasterlagen gebruiken

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *Details laag*
- *Renderer*
  - *Enkelbands rasters*
  - *Multiband rasters*
- *Waarden bevragen*

De codesnippers op deze pagina hebben de volgende import nodig als u buiten de console van PyQGIS bent:

```
from qgis.core import (
  QgsRasterLayer,
  QgsColorRampShader,
  QgsSingleBandPseudoColorRenderer
)
```

4.1 Details laag

Een rasterlaag bestaat uit één of meer rasterbanden — verwezen als een enkelbands en een multibands raster. Een band vertegenwoordigt een matrix van waarden. Een kleurenafbeelding (bijv. luchtfoto) is een raster bestaande uit rode, blauwe en groene banden. Rasters met één enkele band vertegenwoordigen meestal ofwel doorlopende variabelen (bijv. hoogte) of afzonderlijke variabelen (bijv. landgebruik). In sommige gevallen heeft een rasterlaag een palet en verwijzen waarden in het raster naar de kleuren die zijn opgeslagen in het palet:

De volgende code gaat er van uit dat `rlayer` een object `QgsRasterLayer` is.

```
rlayer = QgsProject.instance().mapLayersByName('srtm')[0]
# get the resolution of the raster in layer unit
```

```
rlayer.width(), rlayer.height()
(919, 619)
# get the extent of the layer as QgsRectangle
rlayer.extent()
<QgsRectangle: 20.06856808199999875 -34.27001076999999896, 20.83945284300000012 -
↪33.75077500700000144>
# get the extent of the layer as Strings
rlayer.extent().toString()
'20.0685680819999988,-34.2700107699999990 : 20.8394528430000001,-33.
↪7507750070000014'
# get the raster type: 0 = GrayOrUndefined (single band), 1 = Palette (single␣
↪band), 2 = Multiband
rlayer.rasterType()
0
# get the total band count of the raster
rlayer.bandCount()
1
# get all the available metadata as a QgsLayerMetadata object
rlayer.metadata()
'<qgis._core.QgsLayerMetadata object at 0x13711d558>'
```

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.

De huidige renderer bevragen:

```
rlayer.renderer()
<qgis._core.QgsSingleBandGrayRenderer object at 0x7f471c1da8a0>
rlayer.renderer().type()
'singlebandgray'
```

Gebruik de methode `setRenderer` van `QgsRasterLayer` om een renderer in te stellen. Er zijn verscheidene klassen voor renderer beschikbaar (afgeleid van `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 aan de waarden. Enkelbands rasters met een palet kunnen ook worden getekend met het 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 het tekenen te gebruiken.

4.2.1 Enkelbands rasters

Laten we zeggen dat we een enkelbands rasterlaag willen renderen met kleuren die variëren van groen naar geel (overeenkomende met pixelwaarden van 0 tot en met 255). In de eerste stap zullen we een object `QgsRasterShader` voorbereiden en de functie shader daarvan configureren:

```
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 pixelwaarden en geassocieerde kleuren. Er zijn drie modi voor interpolatie:

- linear (`Interpolated`): de kleur wordt lineair geïnterpoleerd uit de items van de kleurenkaart boven en onder de pixelwaarde
- discrete (`Discrete`): de kleur wordt genomen uit het dichtstbijzijnde item voor de kleurenkaart met gelijke of hogere waarde
- exact (`Exact`): de kleur wordt niet geïnterpoleerd, alleen pixels met waarden gelijk aan die van de kleurenkaart zullen worden getekend

In de tweede stap zullen we deze shader associeren met de rasterlaag:

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

Het getal `1` in de code hierboven is het nummer van de band (rasterbanden worden geïndexeerd vanaf één).

Tenslotte moeten we de methode `triggerRepaint` gebruiken om de resultaten te kunnen zien:

```
rlayer.triggerRepaint()
```

4.2.2 Multiband rasters

Standaard brengt QGIS de eerste drie banden in kaart naar rood, groen en blauw om een kleurenafbeelding te maken (dit is de tekenstijl `MultiBandColor`. In sommige gevallen zou u deze instellen willen overschrijven. De volgende code verwisselt de rode band (1) met de groene band (2):

```
rlayer_multi = QgsProject.instance().mapLayersByName('multiband')[0]
rlayer_multi.renderer().setGreenBand(1)
rlayer_multi.renderer().setRedBand(2)
```

In het geval dat slechts één band nodig is voor de visualisatie van het raster, kan het tekenen van ene enkele band worden gekozen, ofwel grijswaarden of pseudokleur.

We moeten de methode `triggerRepaint` gebruiken om de kaart bij te werken en de resultaten te zien:

```
rlayer_multi.triggerRepaint()
```

4.3 Waarden bevragen

Rasterwaarden kunnen worden bevraagd met de methode `sample` van de klasse `QgsRasterDataProvider`. U dient een `QgsPointXY` te specificeren en het bandnummer van de rasterlaag die u wilt bevragen. De methode geeft een tuple terug met de waarde en `True` of `False`, afhankelijk van de resultaten:

```
val, res = rlayer.dataProvider().sample(QgsPointXY(20.50, -34), 1)
```

Een andere methode om rasterwaarden te bevragen is met de methode `identify()` die een object `QgsRasterIdentifyResult` teruggeeft.

```
ident = rlayer.dataProvider().identify(QgsPointXY(20.5, -34), QgsRaster.
↪IdentifyFormatValue)
```

```
if ident.isValid():
  print(ident.results())
```

In dit geval geeft de methode `results` een woordenboek terug, met indices van banden als sleutels, en bandwaarden als waarden. Bijvoorbeeld iets als `{1:  323.0}`

CHAPTER 5

Vectorlagen gebruiken

- *Informatie over attributen ophalen*
- *Itereren over vectorlagen*
- *Objecten selecteren*
  - *Toegang tot attributen*
  - *Itereren over geselecteerde objecten*
  - *Itereren over een deel van de objecten*
- *Vectorlagen bewerken*
  - *Objecten toevoegen*
  - *Objecten verwijderen*
  - *Objecten bewerken*
  - *Vectorlagen bewerken met een bewerkingsbuffer*
  - *Velden toevoegen en verwijderen*
- *Ruimtelijke index gebruiken*
- *Vectorlagen maken*
  - *Vanuit een instance van `QgsVectorFileWriter`*
  - *Direct uit objecten*
  - *Vanuit een instance van `QgsVectorLayer`*
- *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.

Het meeste werk hier is gebaseerd op de methoden van de klasse `QgsVectorLayer`.

5.1 Informatie over attributen ophalen

U kunt informatie ophalen over de velden die zijn geassocieerd met een vectorlaag door `fields()` aan te roepen op een object `QgsVectorLayer`:

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

5.2 Itereren over vectorlagen

Het doorlopen van de objecten in een vectorlaag is één van de meest voorkomende taken. Hieronder staat een voorbeeld van eenvoudige basiscode om deze taak uit te voeren en enige informatie weer te geven over elk object. Voor de variabele `layer` wordt aangenomen dat die een object `QgsVectorLayer` heeft

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

for feature in features:
    # retrieve every feature with its geometry and attributes
    print("Feature ID: ", feature.id())
    # fetch geometry
    # show some information about the feature geometry
    geom = feature.geometry()
    geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
    if geom.type() == QgsWkbTypes.PointGeometry:
        # the geometry type can be of single or multi type
        if geomSingleType:
            x = geom.asPoint()
            print("Point: ", x)
        else:
            x = geom.asMultiPoint()
            print("MultiPoint: ", x)
    elif geom.type() == QgsWkbTypes.LineGeometry:
        if geomSingleType:
            x = geom.asPolyline()
            print("Line: ", x, "length: ", geom.length())
        else:
            x = geom.asMultiPolyline()
            print("MultiLine: ", x, "length: ", geom.length())
    elif geom.type() == QgsWkbTypes.PolygonGeometry:
        if geomSingleType:
            x = geom.asPolygon()
            print("Polygon: ", x, "Area: ", geom.area())
        else:
            x = geom.asMultiPolygon()
            print("MultiPolygon: ", x, "Area: ", geom.area())
    else:
```

```
        print("Unknown or invalid geometry")
    # fetch attributes
    attrs = feature.attributes()
    # attrs is a list. It contains all the attribute values of this feature
    print(attrs)
```

5.3 Objecten selecteren

In QGIS desktop kunnen objecten op verschillende manieren worden geselecteerd, de gebruiker kan klikken op een object, een rechthoek in het kaartvenster tekenen of een expressie-filter gebruiken. Geselecteerde objecten worden normaal gesproken geaccentueerd in een andere kleur (standaard is geel) om de aandacht van de gebruiker naar de selectie te trekken.

Sometimes it can be useful to programmatically select features or to change the default color.

De methode `selectAll()` kan worden gebruikt om alle objecten te selecteren:

```
# Get the active layer (must be a vector layer)
layer = iface.activeLayer()
layer.selectAll()
```

Gebruik de methode `selectByExpression()` om te selecteren met een expressie:

```
# Assumes that the active layer is points.shp file from the QGIS test suite
# (Class (string) and Heading (number) are attributes in points.shp)
layer = iface.activeLayer()
layer.selectByExpression('"Class"=\'B52\' and "Heading" > 10 and "Heading" <70',␣
↪QgsVectorLayer.SetSelection)
```

U kunt, om de kleur van de selectie te wijzigen, de methode `setSelectionColor()` van `QgsMapCanvas` gebruiken, zoals weergegeven in het volgende voorbeeld:

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

U kunt, om objecten toe te voegen aan de lijst met geselecteerde objecten voor een bepaalde laag, `select()` aanroepen, die de lijst met ID's voor de objecten doorgeeft aan de lijst:

```
selected_fid = []

# Get the first feature id from the layer
for feature in layer.getFeatures():
    selected_fid.append(feature.id())
    break

# Add these features to the selected list
layer.select(selected_fid)
```

De selectie opheffen:

```
layer.removeSelection()
```

5.3.1 Toegang tot attributen

Naar attributen kan worden verwezen door middel van hun naam:

```
print(feature['name'])
```

Als alternatief kan naar attributen worden verwezen door middel van een index. Dit is iets sneller dan het gebruiken van de naam. Bijvoorbeeld om het eerste attribuut te krijgen:

```
print(feature[0])
```

5.3.2 Itereren over geselecteerde objecten

Als u alleen geselecteerde objecten nodig hebt, kunt u de methode `selectedFeatures()` gebruiken van de vectorlaag:

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

5.3.3 Itereren over een deel van de objecten

Wanneer u een deel van de objecten in een laag wilt doorlopen, zoals bijvoorbeeld alleen de objecten in een opgegeven gebied, dan dient een object `QgsFeatureRequest` te worden toegevoegd aan de aanroep `getFeatures()`. Hier is een voorbeeld:

```
areaOfInterest = QgsRectangle(450290,400520, 450750,400780)

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

Omwille van de snelheid wordt het kruisen vaak gedaan door alleen het begrenzingsvak van het object te gebruiken. Er is echter een vlag `ExactIntersect` dat er voor zorgt dat alleen kruisende objecten zullen worden teruggegeven:

```
request = QgsFeatureRequest().setFilterRect(areaOfInterest).
↪setFlags(QgsFeatureRequest.ExactIntersect)
```

Met `setLimit()` kunt u het aantal gezochte objecten beperken. Hier is een voorbeeld:

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

Als u in plaats daarvan een op attributen gebaseerd filter nodig heeft (of als aanvulling) van een ruimtelijke zoals weergegeven in de voorbeelden hierboven, kunt u een object `QgsExpression` bouwen en dat doorgeven aan de constructor `QgsFeatureRequest`. Hier is een voorbeeld:

```
# 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)
```

Bekijk *Expressies, filteren en waarden berekenen* voor de details over de door `QgsExpression` ondersteunde syntaxis.

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 to increase the "speed" of the request
request.setSubsetOfAttributes([0,2])

# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.fields())

# Don't return geometry objects to increase the "speed" of the request
request.setFlags(QgsFeatureRequest.NoGeometry)

# Fetch only the feature with id 45
request.setFilterFid(45)

# The options may be chained
request.setFilterRect(areaOfInterest).setFlags(QgsFeatureRequest.NoGeometry).
↪setFilterFid(45).setSubsetOfAttributes([0,2])
```

5.4 Vectorlagen bewerken

De meeste vector gegevensproviders ondersteunen het bewerken van gegevens van de laag. Soms ondersteunen zij slechts een subset van mogelijke acties voor bewerken. Gebruik de functie `capabilities()` om uit te zoeken welke set voor functionaliteiten wordt ondersteund.

```
caps = layer.dataProvider().capabilities()
# Check if a particular capability is supported:
if caps & QgsVectorDataProvider.DeleteFeatures:
    print('The layer supports DeleteFeatures')
```

Bekijk, voor een lijst van alle beschikbare capabilities, de `API Documentation of QgsVectorDataProvider`.

U kunt, om de tekstuele beschrijving van de capabilities van de laag af te drukken naar een kommagescheiden lijst, `capabilitiesString()` gebruiken, zoals in het volgende voorbeeld:

```
caps_string = layer.dataProvider().capabilitiesString()
# Print:
# 'Add Features, Delete Features, Change Attribute Values, Add Attributes,
# Delete Attributes, Rename Attributes, Fast Access to Features at ID,
# Presimplify Geometries, Presimplify Geometries with Validity Check,
# Transactions, Curved Geometries'
```

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.triggerRepaint()
else:
    iface.mapCanvas().refresh()
```

5.4.1 Objecten toevoegen

Maak enkele instances `QgsFeature` en geef daar een lijst van door aan de methode `addFeatures()` van de provider. Het zal twee waarden teruggeven: resultaat (true/false) en een lijst van toegevoegde objecten (hun ID wordt ingesteld door de opslag van de gegevens).

U kunt, om de attributen in te stellen, ofwel het object initialiseren door een object `QgsFields` door te geven (u kunt dat verkrijgen vanuit de methode `fields()` van de vectorlaag) of `initAttributes()` aan te roepen en het aantal velden op te geven die wilt hebben toegevoegd.

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.fields())
    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.fromPointXY(QgsPointXY(123, 456)))
    (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

5.4.2 Objecten verwijderen

Geef eenvoudigweg een lijst van hun object-ID's op om enkele objecten te verwijderen.

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

5.4.3 Objecten bewerken

Het is mogelijk om de geometrie van objecten te wijzigen of enkele attributen. Het volgende voorbeeld wijzigt eerst waarden van attributen met de index 0 en 1, en wijzigt dan de geometrie van het object.

```
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.fromPointXY(QgsPointXY(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

Tip: Voorkeur voor klasse QgsVectorLayerEditUtils voor bewerken van alleen de geometrie

Als u alleen geometrieën wilt wijzigen, kunt u overwegen `QgsVectorLayerEditUtils` te gebruiken wat enkele nuttige methoden verschaft om geometrieën te bewerken (vertalen, invoegen of punten verplaatsen etc.)

5.4.4 Vectorlagen bewerken met een bewerkingsbuffer

Bij het bewerken van vectoren binnen de toepassing QGIS, moet u eerst de modus Bewerken starten voor een bepaalde laag, dan enige aanpassingen te doen en tenslotte de wijzigingen vastleggen (of terugdraaien). Alle aanpassingen die u doet worden niet weggeschreven totdat u ze vastlegt — zij blijven in de bewerkingsbuffer van het geheugen van de laag. Het is mogelijk om deze functionaliteit ook programmatisch te gebruiken — het is simpelweg een andere methode voor het bewerken van vectorlagen die het direct gebruik van providers van gegevens aanvult. Gebruik deze optie bij het verschaffen van enkele gereedschappen voor de GUI voor het bewerken van vectorlagen, omdat dit de gebruiker in staat zal stellen te bepalen om vast te leggen/terug te draaien

en maakt het gebruiken van Ongedaan maken/Opnieuw mogelijk. Bij het vastleggen van wijzigingen worden alle aanpassingen in de bewerkingsbuffer opgeslagen in de provider van de gegevens.

De methoden zijn soortgelijk aan die welke we hebben gezien in de provider, maar zij worden in plaats daarvan aangeroepen op het object `QgsVectorLayer`.

De laag met in de modus Bewerken staan om deze methoden te kunnen laten werken. Gebruikt de methode `startEditing()` om de modus Bewerken te starten. Gebruik de methoden `commitChanges()` of `rollBack()` om het bewerken te stoppen. De eerste zal al uw wijzigingen vastleggen in de gegevensbron, terwijl de tweede ze zal negeren en de gegevensbron in het geheel niet zal wijzigen.

Gebruik de methode `isEditable()` om te weten te komen of een laag in de modus Bewerken staat.

Hier zijn enkele voorbeelden die demonstreren hoe deze methoden voor bewerken te gebruiken.

```
from qgis.PyQt.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 a given value
layer.changeAttributeValue(fid, fieldIndex, value)

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

De hierboven vermelde aanroepen moeten zijn opgenomen in opdrachten Ongedaan maken om er voor te zorgen dat Ongedaan maken/Opnieuw juist werkt. (Als Ongedaan maken/Opnieuw voor u niet van belang is en u wilt dat de wijzigingen onmiddellijk worden opgeslagen, dan zult u gemakkelijker werken met *bewerken met gegevensprovider*.)

Hier staat hoe u de functionaliteit Ongedaan maken kunt gebruiken:

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

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

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

De methode `beginEditCommand()` zal een interne "actieve" opdracht maken en zal opvolgende wijzigingen in de vectorlaag opnemen. Met de aanroep naar `endEditCommand()` wordt de opdracht doorgegeven aan de stapel Ongedaan maken en de gebruiker zal in staat zijn om Ongedaan maken/Opnieuw uit te voeren vanuit de GUI. Voor het geval er iets verkeerd gaat bij het maken van de wijzigingen, zal de methode `destroyEditCommand()` de opdracht verwijderen en de wijzigingen terugdraaien die al werden gemaakt toen deze opdracht actief was.

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 = next(layer.getFeatures())
```

```
feat[0] = 5
layer.updateFeature(feat)
```

Dit zal aan het einde automatisch `commitChanges()` aanroepen. Indien er een uitzondering optreedt, zal het `rollBack()` alle wijzigingen. In het geval dat een probleem wordt tegengekomen binnen `commitChanges()` (als de methode False teruggeeft) zal een uitzondering `QgsEditError` optreden.

5.4.5 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 qgis.PyQt.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()
```

Tip: Wijzigingen direct opslaan met op `with` gebaseerde opdracht

Gebruiken van `with edit(layer):` de wijzigingen zullen automatisch worden vastgelegd door het aanroepen van `commitChanges()` aan het einde. Indien er een uitzondering optreedt, zal het `rollBack()` alle wijzigingen.. Bekijk *Vectorlagen bewerken met een bewerkingsbuffer*.

5.5 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:

- maak een ruimtelijke index met de klasse `QgsSpatialIndex()`:

  ```
  index = QgsSpatialIndex()
  ```

- voeg objecten aan de index toe — index neemt object `QgsFeature` en voegt dat toe aan de interne gegevensstructuur. U kunt het object handmatig maken of er een gebruiken uit een eerdere aanroep naar `getFeatures()` van de provider.

```
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(QgsPointXY(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))
```

5.6 Vectorlagen maken

Er zijn verschillende manieren om een gegevensset uit een vectorlaag te maken:

- de klasse `QgsVectorFileWriter`: Een handige klasse voor het schrijven van vectorbestanden naar schijf, ofwel met een statische aanroep naar `writeAsVectorFormat()` die de gehele vectorlaag opslaat of een instance van de klasse maken en aanroepen uitvoeren naar `addFeature()`. Deze klasse ondersteunt alle indelingen voor vector die OGR ondersteunt (GeoPackage, Shapefile, GeoJSON, KML en andere).
- de klasse `QgsVectorLayer`: instantieert een gegevensprovider die het opgegeven pad (URL) van de gegevensbron interpreteert om te verbinden met en toegang te verschaffen tot de gegevens. Het kan worden gebruikt om tijdelijke, op geheugen gebaseerde lagen (`memory`), te maken en te verbinden met gegevenssets van OGR (`ogr`), databases (`postgres`, `spatialite`, `mysql`, `mssql`) en meer (`wfs`, `gpx`, `delimitedtext`...).

5.6.1 Vanuit een instance van `QgsVectorFileWriter`

```
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormat(layer,
                                                "/path/to/folder/my_data",
                                                "")
if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
```

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
error = QgsVectorFileWriter.writeAsVectorFormat(layer,
                                                "/path/to/folder/my_esridata",
                                                "UTF-8",
                                                driverName="ESRI Shapefile")
if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
```

De derde (verplichte) parameter specificeert de codering voor de uit te voeren tekst. Slechts enkele stuurprogramma's hebben dit voor een juiste verwerking nodig - Shapefile is er daar één van (andere stuurprogramma's zullen deze parameter negeren). Specificeren van de juiste codering is alleen belangrijk als u internationale tekens (niet US-ASCII) gebruikt.

```
# Write to an ESRI GDB file
opts = QgsVectorFileWriter.SaveVectorOptions()
opts.driverName = "FileGDB"
# if no geometry
opts.overrideGeometryType = QgsWkbTypes.NullGeometry
```

```
opts.actionOnExistingFile = QgsVectorFileWriter.ActionOnExistingFile.
↪CreateOrOverwriteLayer
opts.layerName = 'my_new_layer_name'
error = QgsVectorFileWriter.writeAsVectorFormat(layer=vlayer,
                                                fileName=gdb_path,
                                                options=opts)
if error[0] == QgsVectorFileWriter.NoError:
  print("success!")
else:
  print(error)
```

U kunt ook velden converteren om ze uitwisselbaar te maken met verschillende indelingen door de klasse `FieldValueConverter` te gebruiken. Bijvoorbeeld om typen arryvariabelen (bijv. in Postgres) te converteren naar een type tekst, kunt u het volgende doen:

```
LIST_FIELD_NAME = 'xxxx'

class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

  def __init__(self, layer, list_field):
    QgsVectorFileWriter.FieldValueConverter.__init__(self)
      self.layer = layer
      self.list_field_idx = self.layer.fields().indexFromName(list_field)

  def convert(self, fieldIdxInLayer, value):
    if fieldIdxInLayer == self.list_field_idx:
      return QgsListFieldFormatter().representValue(layer=vlayer,
                                                    fieldIndex=self.list_field_idx,
                                                    config={},
                                                    cache=None,
                                                    value=value)
    else:
      return value

  def fieldDefinition(self, field):
    idx = self.layer.fields().indexFromName(field.name())
    if idx == self.list_field_idx:
      return QgsField(LIST_FIELD_NAME, QVariant.String)
    else:
      return self.layer.fields()[idx]

converter = ESRIValueConverter(vlayer, LIST_FIELD_NAME)
#opts is a QgsVectorFileWriter.SaveVectorOptions as above
opts.fieldValueConverter = converter
```

Een doel-CRS mag ook worden gespecificeerd — als een geldige instance van `QgsCoordinateReferenceSystem` wordt doorgegeven als de vierde parameter, wordt de laag naar dat CRS getransformeerd.

Roep voor geldige namen van stuurprogramma's de methode `supportedFiltersAndFormats` aan of raadpleeg de door OGR ondersteunde indelingen — u zou de waarde in de kolom "Code" moeten doorgeven als de naam van het stuurprogramma.

Optioneel kunt u instellen of of u alleen geselecteerde objecten wilt exporteren, meer driver-specifieke opties voor maken wilt doorgeven of de schrijven wilt vertellen om geen attributen aan te maken... Er zijn een aantal andere (optionele) parameters; bekijk de documentatie voor `QgsVectorFileWriter` voor details.

5.6.2 Direct uit objecten

```
from qgis.PyQt.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", "UTF-8", fields, QgsWkbTypes.Point,␣
↪driverName="ESRI Shapefile")

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

# add a feature
fet = QgsFeature()

fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk
del writer
```

5.6.3 Vanuit een instance van `QgsVectorLayer`

Laten we, naast alle gegevensproviders die worden ondersteund door de klasse `QgsVectorLayer`, ons focussen op de op geheugen gebaseerde lagen. Memory-provider is bedoeld om hoofdzakelijk te worden gebruikt door plug-ins of ontwikkelaars voor 3e partijen. Het slaat geen gegevens op de schijf op, wat ontwikkelaars in staat stelt het te gebruiken als snel backend voor enkele tijdelijke lagen.

De provider ondersteunt velden string, int en double.

De memory-provider ondersteunt ook ruimtelijke indexen, wat wordt ingeschakeld door de functie van de provider `createSpatialIndex()` aan te roepen. Als de ruimtelijke index eenmaal is gemaakt zult u in staat zijn objecten in kleinere regio's sneller te doorlopen (omdat het niet nodig is door alle objecten te gaan, alleen die in de gespecificeerde rechthoek).

Een memory-provider wordt gemaakt door `"memory"` door te geven als de string voor de provider string aan de constructor `QgsVectorLayer`.

De constructor accepteert ook een URI die het type geometrie van de laag definieert, één van: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, `"MultiPolygon"` of `"None"`.

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

crs=definition Specificeert het coördinaten referentiesysteem, waar definition een van de vormen kan zijn die worden geaccepteerd door `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 qgis.PyQt.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.fromPointXY(QgsPointXY(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 = vl.extent()
print("extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum())

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

5.7 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.

De renderer voor een bepaalde laag kan worden verkregen zoals hieronder is weergegeven:

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

En met die verwijzing, laten we het een beetje verkennen

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

Er zijn verschillende bekende typen renderer beschikbaar in de bron-bibliotheek van QGIS:

Type	Klasse	Omschrijving
singleSymbol	`QgsSingleSymbolRenderer`	Rendert alle objecten met hetzelfde symbool
categorizedSymbol	`QgsCategorizedSymbolRenderer`	Rendert objecten door een ander symbool voor elke categorie te gebruiken
graduatedSymbol	`QgsGraduatedSymbolRenderer`	Rendert objecten door een ander symbool voor elke bereik van waarden te gebruiken

Er kunnen ook enkele aangepaste typen renderer zijn, dus doe nooit de aanname dat alleen deze typen beschikbaar zijn. U kunt het `QgsRendererRegistry` van de toepassing bevragen om de huidige beschikbare renderers te achterhalen:

```
print(QgsApplication.rendererRegistry().renderersList())
# Print:
['nullSymbol',
'singleSymbol',
'categorizedSymbol',
'graduatedSymbol',
'RuleRenderer',
'pointDisplacement',
'pointCluster',
'invertedPolygonRenderer',
'heatmapRenderer',
'25dRenderer']
```

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(renderer.dump())
```

5.7.1 Renderer Enkel symbool

U kunt het voor de rendering gebruikte symbool verkrijgen door de methode `symbol()` aan te roepen en die te wijzigen met de methode `setSymbol()` (opmerking voor ontwikkelaars in C++: de renderer wordt eigenaar van het symbool.)

U kunt het symbool dat wordt gebruikt door een bepaalde vectorlaag wijzigen door `setSymbol()` aan te roepen die een instance doorgeeft van de toepasselijke symbool instance. Symbolen voor lagen *punt*, *lijn* en *polygoon* kunnen worden gemaakt door het aanroepen van de functie `createSimple()` van de overeenkomende klassen `QgsMarkerSymbol`, `QgsLineSymbol` en `QgsFillSymbol`.

Het aan `createSimple()` doorgegeven woordenboek stelt de eigenschappen voor de stijl van het symbool in.

U kunt bijvoorbeeld het gebruikte symbool voor een bepaalde **punt**-laag wijzigen door `setSymbol()` aan te roepen die een instance doorgeeft van een :class:‘QgsMarkerSymbol <qgis.core.QgsMarkerSymbol> zoals in het volgende voorbeeld van code:

```
symbol = QgsMarkerSymbol.createSimple({'name': 'square', 'color': 'red'})
layer.renderer().setSymbol(symbol)
# show the change
layer.triggerRepaint()
```

`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

U kunt de voorbeeldcode volgen om een volledige lijst met eigenschappen te verkrijgen van de eerste symboollaag van een instance symbool :

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

Dit kan nuttig zijn als u enkele eigenschappen wilt wijzigen:

```
# You can alter a single property...
layer.renderer().symbol().symbolLayer(0).setSize(3)
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.renderer().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.renderer().setSymbol(QgsMarkerSymbol.createSimple(props))
# show the changes
layer.triggerRepaint()
```

5.7.2 Renderer symbool Categoriën

Bij het gebruiken van een renderer Categorieën kunt u het attribuut dat is gebruikt voor de classificatie bevragen en instellen: gebruik de methoden `classAttribute()` en `setClassAttribute()`.

Een lijst categorieën verkrijgen

```
for cat in renderer.categories():
    print("{}: {} :: {}".format(cat.value(), cat.label(), cat.symbol()))
```

Waar `value()` de waarde is die wordt gebruikt voor het onderscheiden van de categorieën, `label()` is een tekst die gebruikt wordt voor de omschrijving van de categorie en de methode `symbol()` geeft het toegewezen symbool terug.

De renderer slaat gewoonlijk ook het originele symbool en de kleurenbalk op die voor de classificatie werden gebruikt: methoden `sourceColorRamp()` en `sourceSymbol()`.

5.7.3 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 renderer.ranges():
    print("{} - {}: {} {}".format(
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
        ran.symbol()
      ))
```

U kunt opnieuw de methoden `classAttribute` (om de naam van het attribuut voor classificatie te zoeken), `sourceSymbol` en `sourceColorRamp` gebruiken. 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.PyQt import QtGui

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 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setOpacity(myOpacity)
myRange1 = QgsRendererRange(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 = QgsSymbol.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setOpacity(myOpacity)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRenderer.EqualInterval)
```

```
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
QgsProject.instance().addMapLayer(myVectorLayer)
```

5.7.4 Werken met symbolen

Voor het weergeven van symbolen is er de basisklasse `QgsSymbol` met drie afgeleide klassen:

- `QgsMarkerSymbol` — voor objecten punt
- `QgsLineSymbol` — voor objecten lijn
- `QgsFillSymbol` — voor objecten polygoon

Elk symbool bestaat uit één of meer symboollagen (klassen afgeleid van `QgsSymbolLayer`). De symboollagen doen de actuele rendering, de symboolklasse zelf dient alleen als een container voor de symboollagen.

Met een instance van een symbool (bijv. van een renderer), is het mogelijk om het te verkennen: de methode `type` zegt of het een symbool markering, lijn of vulling is. Er is de methode `dump` wat een korte omschrijving van het symbool teruggeeft. Een lijst van symboollagen verkrijgen:

```
for i in range(symbol.symbolLayerCount()):
    lyr = symbol.symbolLayer(i)
    print("{}: {}".format(i, lyr.layerType()))
```

Gebruik de methode `color` om de kleur van het symbool vast te stellen en `setColor` en `angle`, voor lijnsymbolen geeft de methode `width` de dikte van de lijn terug.

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

Werken met symboollagen

Zoals eerder gezegd bepalen symboollagen (subklassen van `QgsSymbolLayer`) het uiterlijk van de objecten. Er zijn verscheidene basisklassen voor symboollagen voor algemeen gebruik. Het is mogelijk om nieuwe typen symboollagen te implementeren en dus willekeurig aan te passen hoe objecten zullen worden gerenderd. De methode `layerType()` identificeert uniek de klasse van de symboollaag — de basis en standaard zijn de typen symboollagen `SimpleMarker`, `SimpleLine` en `SimpleFill`.

U kunt een volledige lijst van de typen symboollagen, die u voor een bepaalde klasse van een symboollaag kunt maken, verkrijgen met de volgende code:

```
from qgis.core import QgsSymbolLayerRegistry
myRegistry = QgsApplication.symbolLayerRegistry()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbol.Marker):
    print(item)
```

Uitvoer:

```
EllipseMarker
FilledMarker
FontMarker
GeometryGenerator
SimpleMarker
SvgMarker
VectorField
```

Klasse `QgsSymbolLayerRegistry` beheert een database van alle beschikbare typen symboollagen.

Gebruik zijn methode `properties()` om toegang te verkrijgen tot de gegevens van de symboollaag, die een woordenboek met paren van sleutels-waarden teruggeeft van eigenschappen die het uiterlijk bepalen. Elke type

symboollaag heeft een specifieke set eigenschappen die het gebruikt. Aanvullend zijn er de generieke methoden `color`, `size`, `angle` en `width` met hun tegenhangers om ze in te stellen. Natuurlijk zijn grootte en hoek alleen beschikbaar voor symboollagen voor markeringen en breedte voor lijn-symboollagen.

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

```
from qgis.core import QgsMarkerSymbolLayer
from qgis.PyQt.QtGui import QColor

class FooSymbolLayer(QgsMarkerSymbolLayer):

  def __init__(self, radius=4.0):
      QgsMarkerSymbolLayer.__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)
```

De methode `layerType` bepaalt de naam van de symboollaag, die moet uniek zijn voor alle symboollagen. De methode `properties` wordt gebruikt voor het behouden van attributen. De methode `clone` moet een kopie teruggeven van de symboollaag met exact dezelfde attributen. Tenslotte zijn er methoden voor renderen: `startRender` wordt aangeroepen vóór het renderen van het eerste object, `stopRender` als het renderen is voltooid en de methode `renderPoint` wordt aangeroepen om het renderen uit te voeren. De coördinaten van de punt(en) zijn al getransformeerd naar de coördinaten voor uitvoer.

Voor polylijnen en polygonen zou het enige verschil liggen in de methode van renderen: u zou `renderPolyline` gebruiken, welke een lijst met lijnen zou ontvangen, terwijl `renderPolygon` een lijst van punten op de buitenste ring als de eerste parameter ontvangt en een lijst van binnenringen (of None) als een tweede 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

```
from qgis.gui import QgsSymbolLayerWidget
```

```
class FooSymbolLayerWidget(QgsSymbolLayerWidget):
    def __init__(self, parent=None):
        QgsSymbolLayerWidget.__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 methode `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 het voorbeeld 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

```
from qgis.core import QgsSymbol, QgsSymbolLayerAbstractMetadata,␣
↪QgsSymbolLayerRegistry

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

  def __init__(self):
    QgsSymbolLayerAbstractMetadata.__init__(self, "FooMarker", QgsSymbol.Marker)

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

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

```
QgsApplication.symbolLayerRegistry().addSymbolLayerType(FooSymbolLayerMetadata())
```

U zou het type laag (hetzelfde als welke wordt teruggegeven door de laag) en type symbool (markering/lijn/vulling) moeten doorgeven aan de constructor van de bovenliggende klasse. De methode `createSymbolLayer()` zorgt voor het maken van een instance van de symboollaag met attributen die zijn gespecificeerd in het woordenboek *props*. En er is de methode `createSymbolLayerWidget()` die de instellingen voor de widget teruggeeft voor dit type symboollaag.

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

5.7.5 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
from qgis.core import QgsWkbTypes, QgsSymbol, QgsFeatureRenderer


class RandomRenderer(QgsFeatureRenderer):
  def __init__(self, syms=None):
    QgsFeatureRenderer.__init__(self, "RandomRenderer")
    self.syms = syms if syms else [QgsSymbol.defaultSymbol(QgsWkbTypes.
↪geometryType(QgsWkbTypes.Point))]

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

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

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

  def usedAttributes(self):
    return []

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

from qgis.gui import QgsRendererWidget
class RandomRendererWidget(QgsRendererWidget):
  def __init__(self, layer, style, renderer):
    QgsRendererWidget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButton()
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
```

```
        self.setLayout(self.vbox)
        self.btn1.clicked.connect(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
```

De constructor van de bovenliggende klasse `QgsFeatureRenderer` heeft de naam van de renderer nodig (die uniek moet zijn voor alle renderers). De methode `symbolForFeature` is die welke bepaalt welk symbool zal worden gebruikt voor een bepaald object. `startRender` en `stopRender` zorgen voor initialisatie/finalisatie van het renderen van het symbool. De methode `usedAttributes` kan een lijst met veldnamen teruggeven waarvan de renderer verwacht dat die aanwezig is. Tenslotte zou de methode `clone` een kopie van de renderer moeten teruggeven.

Net als met symboollagen is het mogelijk een GUI toe te voegen voor de configuratie van de renderer. Die moet worden afgeleid uit `QgsRendererWidget`. De volgende voorbeeldcode maakt een knop die de gebruiker in staat stelt het eerste symbool in te stellen

```
from qgis.gui import QgsRendererWidget, QgsColorButton

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

De constructor ontvangt instances van de actieve laag (`QgsVectorLayer`), de globale opmaak (`QgsStyle`) en huidige renderer. Indien er geen renderer is of de renderer heeft een andere type, zal die worden vervangen door onze nieuwe renderer, anders zullen we de huidige renderer gebruiken (die al het type heeft dat we nodig hebben). De inhoud van de widget zou moeten worden bijgewerkt om de huidige staat van de renderer weer te geven. Wanneer het dialoogvenster van de renderer wordt geaccepteerd, wordt de methode voor de widget `renderer` aangeroepen om de huidige renderer te verkrijgen — die zal worden toegewezen aan de laag.

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

```
from qgis.core import QgsRendererAbstractMetadata,QgsRendererRegistry,
↪QgsApplication
```

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

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

QgsApplication.rendererRegistry().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.

U kunt, om een pictogram te associëren met de renderer, die toewijzen in de constructor `QgsRendererAbstractMetadata` als een derde (optioneel) argument — de basis klassse-constructor in de functie `__init__()` van de RandomRendererMetadata wordt

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

Het pictogram kan ook op een later tijdstip worden geassocieerd met de methode `setIcon` van de klasse van de metadata. Het pictogram kan worden geladen vanuit een bestand (zoals hierboven weergegeven) of kan worden geladen vanuit een Qt resource (PyQt5 bevat .qrc compiler voor Python).

5.8 Meer onderwerpen

TODO:

- symbolen maken/aanpassen
- werken met stijl (`QgsStyle`)
- werken met kleurverlopen (`QgsColorRamp`)
- symboollaag en registraties van renderer verkennen

CHAPTER 6

Afhandeling van geometrie

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

De codesnippers op deze pagina hebben de volgende import nodig als u buiten de console van PyQGIS bent:

```
from qgis.core import (
  QgsGeometry,
  QgsPoint,
  QgsPointXY,
  QgsWkbTypes,
  QgsProject,
  QgsFeatureRequest,
  QgsDistanceArea
)
```

Naar punten, lijnen en polygonen die een ruimtelijk object weergeven wordt gewoonlijk verwezen als geometrieën. In QGIS worden zij weergegeven door de klasse `QgsGeometry`.

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.

Beschrijving en specificaties van alle mogelijke constructies van geometrieën en relaties zijn beschikbaar in de OGC Simple Feature Access Standards voor uitgebreide details.

6.1 Construeren van geometrie

PyQGIS verschaft verscheidene opties voor het maken van een geometrie:

- uit coördinaten

```
gPnt = QgsGeometry.fromPointXY(QgsPointXY(1,1))
print(gPnt)
gLine = QgsGeometry.fromPolyline([QgsPoint(1, 1), QgsPoint(2, 2)])
print(gLine)
gPolygon = QgsGeometry.fromPolygonXY([[QgsPointXY(1, 1),
    QgsPointXY(2, 2), QgsPointXY(2, 1)]])
print(gPolygon)
```

Coördinaten worden opgegeven met behulp van de klassen `QgsPoint` of `QgsPointXY`. Het verschil tussen deze klassen is dat `QgsPoint` dimensies M en Z ondersteunt.

Een Polylijn (Lijn) wordt weergegeven door een lijst met punten.

Een polygoon wordt weergegeven als een lijst van lineaire ringen (d.i. gesloten lijnen). De eerste ring is de buitenste ring (grens), optionele volgende ringen zijn gaten in de polygoon. Onthoud dat, anders dan andere programma's, QGIS de ring voor u zal sluiten dus is er geen reden om het eerste punt als laatste te dupliceren.

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)

```
geom = QgsGeometry.fromWkt("POINT(3 4)")
print(geom)
```

- uit bekende binaire (WKB)

```
g = QgsGeometry()
wkb = bytes.fromhex("010100000000000000000045400000000000001440")
g.fromWkb(wkb)

# print WKT representation of the geometry
print(g.asWkt())
```

6.2 Toegang tot geometrie

Als eerste zou u het type geometrie moeten zoeken, de methode `wkbType()` is die om te gebruiken. Het geeft een waarde uit de enumeratie `QgsWkbTypes.Type` terug.

```
if gPnt.wkbType() == QgsWkbTypes.Point:
  print(gPnt.wkbType())
  # output: 1 for Point
if gLine.wkbType() == QgsWkbTypes.LineString:
  print(gLine.wkbType())
if gPolygon.wkbType() == QgsWkbTypes.Polygon:
  print(gPolygon.wkbType())
  # output: 3 for Polygon
```

Als alternatief kan men de methode `type()` gebruiken die een waarde teruggeeft uit de enumeratie van de klasse `QgsWkbTypes.GeometryType`.

U kunt de functie `displayString()` gebruiken om een voor mensen leesbaar type geometrie te verkrijgen.

```
print(QgsWkbTypes.displayString(gPnt.wkbType()))
# output: 'Point'
print(QgsWkbTypes.displayString(gLine.wkbType()))
# output: 'LineString'
print(QgsWkbTypes.displayString(gPolygon.wkbType()))
# output: 'Polygon'
```

```
Point
LineString
Polygon
```

Er is ook een hulpfunctie `isMultipart()` om uit te zoeken of een geometrie meerdelig is of niet.

Voor elk type vector zijn er functies voor toegang om informatie uit de geometrie op te halen. Hier is een voorbeeld hoe deze functies te gebruiken:

```
print(gPnt.asPoint())
# output: <QgsPointXY: POINT(1 1)>
print(gLine.asPolyline())
# output: [<QgsPointXY: POINT(1 1)>, <QgsPointXY: POINT(2 2)>]
print(gPolygon.asPolygon())
# output: [[<QgsPointXY: POINT(1 1)>, <QgsPointXY: POINT(2 2)>, <QgsPointXY:
↪POINT(2 1)>, <QgsPointXY: POINT(1 1)>]]
```

Notitie: De tuples (x,y) zijn geen echte tuples, zij zijn objecten `QgsPoint`, de waarden zijn toegankelijk met de methoden `x()` en `y()`.

Voor meerdelige geometrieën zijn er soortgelijke functies voor toegang: `asMultiPoint()`, `asMultiPolyline()` en `asMultiPolygon()`.

6.3 Predicaten en bewerking voor geometrieën

QGIS gebruikt de bibliotheek GEOS voor geavanceerde bewerkingen met geometrieën, zoals de predicaten voor geometrieën (`contains()`, `intersects()`, ...) en het instellen van bewerkingen (`combine()`, `difference()`, ...). Het kan ook geometrische eigenschappen van geometrieën berekenen, zoals gebied (in het geval van polygonen) of lengten (voor polygonen en lijnen).

Laten we een voorbeeld bekijken dat het doorlopen van de objecten op een laag combineert met het uitvoeren van enkele geometrische berekeningen, gebaseerd op hun geometrieën. De onderstaande code zal het gebied en de perimeter van elk land op de laag `countries` in ons project in de handleiding voor QGIS berekenen en afdrukken.

De volgende code gaat ervan uit dat `layer` een object `QgsVectorLayer` is.

```
# let's access the 'countries' layer
layer = QgsProject.instance().mapLayersByName('countries')[0]

# let's filter for countries that begin with Z, then get their features
query = '"name" LIKE \'Z%\''
features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))

# now loop through the features, perform geometry computation and print the results
for f in features:
  geom = f.geometry()
  name = f.attribute('NAME')
  print(name)
  print('Area: ', geom.area())
  print('Perimeter: ', geom.length())
```

Nu hebt u de gebieden en perimeters van de geometrieën berekend en afgedrukt. Het zal u echter snel opvallen dat de waarden vreemd zijn. Dat komt omdat gebieden en perimeters geen rekening houden met het CRS bij het berekenen met behulp van de methoden `area()` en `length()` uit de klasse `QgsGeometry`. Voor een meer krachtiger berekening van gebied en afstand kan de klasse `QgsDistanceArea` worden gebruikt, die op ellipsoïde gebaseerde berekeningen kan uitvoeren:

De volgende code gaat ervan uit dat `layer` een object `QgsVectorLayer` is.

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

layer = QgsProject.instance().mapLayersByName('countries')[0]

# let's filter for countries that begin with Z, then get their features
query = '"name" LIKE \'Z%\''
features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))

for f in features:
  geom = f.geometry()
  name = f.attribute('NAME')
  print(name)
  print("Perimeter (m):", d.measurePerimeter(geom))
  print("Area (m2):", d.measureArea(geom))

  # let's calculate and print the area again, but this time in square kilometers
  print("Area (km2):", d.convertAreaMeasurement(d.measureArea(geom), QgsUnitTypes.
↪AreaSquareKilometers))
```

Als alternatief zou u misschien de afstand en richting tussen twee punten willen weten.

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

# Let's create two points.
# Santa claus is a workaholic and needs a summer break,
# lets see how far is Tenerife from his home
santa = QgsPointXY(25.847899, 66.543456)
tenerife = QgsPointXY(-16.5735, 28.0443)

print("Distance in meters: ", d.measureLine(santa, tenerife))
```

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.

- Afstand en gebied gebruiken de klasse `QgsDistanceArea`: Algoritme Afstandsmatrix
- Algoritme Lijnen naar polygonen

CHAPTER 7

Ondersteuning van projecties

- *Coördinaten ReferentieSystemen*
- *CRS transformatie*

Als u buiten de console van PyQGIS bent, hebben de codesnippers op deze pagina de volgende import nodig:

```
from qgis.core import (QgsCoordinateReferenceSystem,
                       QgsCoordinateTransform,
                       QgsProject,
                       QgsPointXY,
                       )
```

7.1 Coördinaten ReferentieSystemen

Coördinaten referentiesystemen (CRS) zijn ingekapseld in de klasse `QgsCoordinateReferenceSystem`. Instances van deze klasse kunnen op verschillende manieren worden gemaakt:

- specificeren van CRS met zijn ID

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

  QGIS gebruikt drie verschillende ID's voor elk referentiesysteem:

  – `InternalCrsId` — ID gebruikt in de interne database van QGIS.

  – `PostgisCrsId` — ID gebruikt in databases van PostGIS.

  – `EpsgCrsId` — ID toegewezen door de organisatie EPSG.

  Indien niet anders gespecificeerd met de 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)
assert crs.isValid()
```

- maak een ongeldig CRS en gebruik dan een van de functies functies `create*` om die te initialiseren. In het volgende voorbeeld gebruiken we een string van Proj4 om de projectie te initialiseren.

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

Het is verstandig om te controleren of het maken (d.i. opzoeken in de database) van het CRS succesvol was: `isValid()` moet `True` teruggeven.

Onthoud dat, voor het initialiseren van ruimtelijke referentiesystemen, QGIS de instellingen moet opzoeken in zijn interne database `srs.db`. Dus dienen bij het maken van een eigen applicatie de paden te worden ingesteld met `QgsApplication.setPrefixPath()` anders zal het vinden van de database mislukken. Als opdrachten worden uitgevoerd vanuit de console van Python in QGIS of vanuit een plug-in hoeft u niets te doen: alles is al goed ingesteld voor u.

Toegang tot informatie ruimtelijk referentiesysteem:

```
crs = QgsCoordinateReferenceSystem(4326)

print("QGIS CRS ID:", crs.srsid())
print("PostGIS SRID:", crs.postgisSrid())
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.isGeographic())
# check type of map units in this CRS (values defined in QGis::units enum)
print("Map units:", crs.mapUnits())
```

Uitvoer:

```
QGIS CRS ID: 3452
PostGIS SRID: 4326
Description: WGS 84
Projection Acronym: longlat
Ellipsoid Acronym: WGS84
Proj4 String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

7.2 CRS transformatie

Het is mogelijk transformaties tussen verschillende ruimtelijke referentiesystemen uit te voeren door gebruik te maken van de klasse `QgsCoordinateTransform`. De eenvoudigste manier om deze functie te gebruiken is een bron en doel CRS te definieren en een instantie van `QgsCoordinateTransform` te construeren (construct) met deze erin en het huidige project. Dan kan de funtie `transform()` herhaaldelijk worden aangeroepen voor het uitvoeren van de transformatie. Standaard wordt van bron naar doel getransformeerd, maar de transformatie kan ook worden omgedraaid.

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

```
xform = QgsCoordinateTransform(crsSrc, crsDest, QgsProject.instance())

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

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

Uitvoer:

```
Transformed point: <QgsPointXY: POINT(832713.79873844375833869 553423.
↪98688333143945783)>
Transformed back: <QgsPointXY: POINT(18 5)>
```

CHAPTER 8

Het kaartvenster gebruiken

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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

De widget Kaartvenster is waarschijnlijk de meest belangrijke widget in QGIS, omdat het de samengestelde kaart weergeeft uit op elkaar gelegde kaartlagen en interactie mogelijk maakt met de kaart en de lagen. Het kaartvenster geeft altijd een gedeelte van de kaart weer, gedefinieerd door het huidige bereik van het kaartvenster. De interactie wordt gedaan door middel van het gebruiken van **gereedschappen voor de kaart**: er zijn gereedschappen pannen, zoomen, identificeren van lagen, meten, bewerken van vector en andere. Soortgelijk aan andere grafische programma's is er altijd één gereedschap actief en de gebruiker kan tussen de verschillende gereedschappen schakelen.

Het kaartvenster wordt geïmplementeerd met de klasse `QgsMapCanvas` in de module `qgis.gui`. De implementatie is gebaseerd op het framework Qt Graphics View. Dat raamwerk verschaft in het algemeen een oppervlak en een weergave waar aangepaste grafische items zijn geplaatst en waarmee de gebruiker interactief kan werken. We gaan er van uit dat u bekend genoeg bent met Qt om de concepten van de grafische scene, weergave en items te begrijpen. Indien niet, zorg er dan voor overview of the framework te hebben gelezen.

Altijd als de kaart is verplaatst, is in-/uitgezoomd (of enkele andere acties die een verversing activeren), wordt de kaart opnieuw gerenderd binnen het huidige bereik. De lagen worden gerenderd naar een afbeelding (met behulp van de klasse `QgsMapRendererJob`) en die afbeelding wordt weergegeven in het kaartvenster. De klasse `QgsMapCanvas` beheert ook het verversen van de gerenderde kaart. Naast dit item, dat optreedt als een achtergrond, kunnen er meer **items voor het kaartvenster** zijn.

Typische items voor het kaartvenster zijn elastieken banden (gebruikt voor meten, bewerken van vectoren etc.) of markeringen van punten. De items voor het kaartvenster worden gewoonlijk gebruikt om een bepaalde visuele

terugkoppeling te geven voor gereedschappen voor de kaart, bijvoorbeeld, bij het maken van een nieuwe polygoon, maakt het gereedschap voor de kaart een item elastieken band die de huidige vorm van de polygoon weergeeft. Alle items voor het kaartvenster zijn sub-klassen van `QgsMapCanvasItem` die iets meer functionaliteit toevoegt aan de basisobjecten `QGraphicsItem`.

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

- kaartvenster — voor het bekijken van de kaart
- items voor het kaartvenster — aanvullende items die kunnen worden weergegeven in het kaartvenster
- gereedschappen voor de kaart — voor interactie met het kaartvenster

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()
```

Dit produceert een zelfstandig venster met een kaartvenster. Het kan ook worden ingebed in een bestaand widget of venster. Plaats een `QWidget` op het formulier en promoveer dat tot een nieuwe klasse: stel `QgsMapCanvas` in als naam voor de klasse en stel `qgis.gui` in als kopbestand. De functionaliteit `pyuic5` zal er zorg voor dragen. Dit is een handige manier om het kaartvenster in te bedden. De andere mogelijkheid is om handmatig de code te schrijven door het kaartvenster en andere widgets (als kinderen van een hoofdvenster of dialoogvenster) te construeren en een lay-out te maken.

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)
```

(Voor het geval u zich dat afvraagt, `Qt` komt van de module `PyQt.QtCore` en `Qt.white` is één van de voorgedefinieerde instanties van `QColor`.)

Nu is het tijd om enkele kaartlagen toe te voegen. We zullen eerst een laag openen en die toevoegen aan het huidige project. Daarna zullen we het bereik van het kaartvenster instellen alsmede de lijst met lagen voor het kaartvenster

```
path_to_ports_layer = os.path.join(QgsProject.instance().homePath(),
                                   "data", "ports", "ports.shp")

vlayer = QgsVectorLayer(path_to_ports_layer, "Ports layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")

# add layer to the registry
QgsProject.instance().addMapLayer(vlayer)

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

# set the map canvas layer set
canvas.setLayers([vlayer])
```

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

8.2 Elastieken banden en markeringen voor punten

Gebruik items voor het kaartvenster om enkele aanvullende gegevens bovenop de kaart in het kaartvenster weer te geven. Het is mogelijk om aangepaste klassen voor items voor het kaartvenster te maken (hieronder behandeld), er zijn voor het gemak echter twee handige klassen voor items voor het kaartvenster: `QgsRubberBand` voor het tekenen van polylijnen of polygonen, en `QgsVertexMarker` voor het tekenen van punten. Zij werken beide met coördinaten op de kaart, dus de vorm wordt automatisch verplaatst/geschaald als het kaartvenster wordt verschoven of als er wordt gezoomd.

Een polylijn weergeven

```
r = QgsRubberBand(canvas, False)  # False = not a polygon
points = [QgsPoint(-100, 45), QgsPoint(10, 60), QgsPoint(120, 45)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Een polygoon weregeven

```
r = QgsRubberBand(canvas, True)  # True = a polygon
points = [[QgsPointXY(-100, 35), QgsPointXY(10, 50), QgsPointXY(120, 35)]]
r.setToGeometry(QgsGeometry.fromPolygonXY(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)
```

De items voor het kaartvenster zijn gebonden aan de scene van het kaartvenster. Gebruik de combinatie `hide()` en `show()` om ze tijdelijk te verbergen (en weer opnieuw weer te geven). U moet het uit de scene van het kaartvenster verwijderen om het item volledig te verwijderen

```
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)

Een elastieken band kan ook gebruikt worden om punten te tekenen, maar de klasse `QgsVertexMarker` is beter geschikt hiervoor (`QgsRubberBand` zou alleen een rechthoek rondom het gewenste punt tekenen).

U kunt de markering voor punten als volgt gebruiken:

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPointXY(10,40))
```

Dit zal een rood kruis tekenen op de positie **[10,45]**. Het is mogelijk om het type pictogram, de grootte, de kleur en de dikte van de pen aan te passen

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

Gebruik dezelfde methode als voor elastieken banden om markeringen voor punten tijdelijk te verbergen en ze uit het kaartvenster te verwijderen.

8.3 Gereedschappen voor de kaart gebruiken in het kaartvenster

Het volgende voorbeeld maakt een venster dat een kaartvenster bevat en basisgereedschappen voor het verschuiven van en zoomen op de kaart. Acties zijn gemaakt voor het activeren van elk gereedschap: verschuiven (pannen) wordt gedaan met `QgsMapToolPan`, in/uitzoomen met een paar instances van `QgsMapToolZoom`. De acties zijn ingesteld als te selecteren en later toegewezen aan het gereedschap om de automatische afhandeling van de status geselecteerd/niet geselecteerd van de acties mogelijk te maken – wanneer een gereedschap voor de kaart wordt geactiveerd, wordt de actie daarvan gemarkeerd als geselecteerd en de actie van het vorige gereedschap voor de kaart wordt gedeselecteerd. De gereedschappen voor de kaart worden geactiveerd met behulp van de methode `setMapTool()`.

```
from qgis.gui import *
from qgis.PyQt.QtWidgets import QAction, QMainWindow
from qgis.PyQt.QtCore import Qt

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.setLayers([layer])

        self.setCentralWidget(self.canvas)

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

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

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

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

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

U kunt bovenstaande code proberen in de bewerker van de console voor Python. Voeg de volgende regels toe om de klasse `MyWnd` te instantiëren om het kaartvenster te activeren. Dat zal de huidige geselecteerde laagr in het nieuw gemaakte kaartvenster renderen

```
w = MyWnd(iface.activeLayer())
w.show()
```

8.4 Aangepaste gereedschappen voor de kaart schrijven

U kunt aangepaste gereedschappen schrijven, om een aagepast gedrag te implementeren voor acties die door gebruikers op het kaartvenster worden uitgevoerd.

Gereedschappen voor de kaart zouden moeten erven van de klasse `QgsMapTool` of een daarvan afgeleide klasse, en in het kaartvenster moeten worden geselecteerd als actief gereedschap met behulp van de methode `setMapTool()` zoals we al eerder hebben gezien.

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, True)
    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(True)

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

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

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

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

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

```
      return

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

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

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

      return QgsRectangle(self.startPoint, self.endPoint)

  def deactivate(self):
    QgsMapTool.deactivate(self)
    self.deactivated.emit()
```

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)
```

CHAPTER 9

Kaart renderen en afdrukken

De codesnippers op deze pagina hebben de volgende import nodig:

```
import os
```


- *Eenvoudig renderen*
- *Lagen met een verschillend CRS renderen*
- *Uitvoer door Afdruklay-out te gebruiken*
  - *Lay-out exporteren*
  - *Een afdrukatlas exporteren*

Er zijn over het algemeen twee benaderingen wanneer ingevoerde gegevens zouden moeten worden gerenderd als een kaart: ofwel doe het op de snelle manier met behulp van *QgsMapRendererJob* of produceer een meer fijn afgestemde uitvoer door de kaart samen te stellen met behulp van de klasse `QgsLayout`.

9.1 Eenvoudig renderen

Het renderen wordt gedaan door een object `QgsMapSettings` te maken om de opties voor renderen te definiëren, en dan een `QgsMapRendererJob` te construeren met deze opties. het laatste wordt dan gebruikt om de resulterende afbeelding te maken.

Hier is een voorbeeld:

```
image_location = os.path.join(QgsProject.instance().homePath(), "render.png")

# e.g. vlayer = iface.activeLayer()
vlayer = QgsProject.instance().mapLayersByName("countries")[0]
options = QgsMapSettings()
options.setLayers([vlayer])
options.setBackgroundColor(QColor(255, 255, 255))
options.setOutputSize(QSize(800, 600))
options.setExtent(vlayer.extent())
```

```
render = QgsMapRendererParallelJob(options)

def finished():
    img = render.renderedImage()
    # save the image; e.g. img.save("/Users/myuser/render.png","png")
    img.save(image_location, "png")
    print("saved")

render.finished.connect(finished)

render.start()
```

9.2 Lagen met een verschillend CRS renderen

Als u meer dan één laag hebt en zij hebben een verschillend CRS, zal het eenvoudige voorbeeld hierboven niet werken: om de juiste waarden uit de berekeningen van het bereik te krijgen dient u expliciet het doel-CRS in te stellen.

```
settings.setLayers(layers)
render.setDestinationCrs(layers[0].crs())
```

9.3 Uitvoer door Afdruklay-out te gebruiken

Afdruklay-out is een zeer handig gereedschap als u een uitgebreidere uitvoer wilt dan de eenvoudige rendering van die welke hierboven is weergegeven. Het is mogelijk complexe lay-outs voor kaarten te maken, bestaande uit weergaven van kaarten, labels, legenda, tabellen en andere elementen die gewoonlijk aanwezig zijn op papieren kaarten. De lay-outs kunnen dan worden geëxporteerd naar PDF, rasterafbeeldingen of direct worden afgedrukt op een printer.

De afdruklay-out bestaat uit een aantal klassen. Zij behoren allemaal tot de bron-bibliotheek. De toepassing QGIS heeft een handige GUI voor het plaatsen van de elementen, hoewel die niet beschikbaar is in de bibliotheek van de GUI. Als u niet bekend bent met het framework Qt Graphics View, dan wordt u aangeraden nu de documentatie te raadplegen, omdat afdruklay-out daarop is gebaseerd.

De centrale klasse van de afdruklay-out is de klasse `QgsLayout` die is afgeleid van de klasse voor Qt QGraphicsScene. Laten we er een instantie van maken:

```
p = QgsProject()
layout = QgsLayout(p)
layout.initializeDefaults()
```

Nu kunnen we verschillende elementen (kaart, label, ...) toevoegen aan de lay-out. Al deze objecten worden weergegeven door klassen die erven van de basisklasse `QgsLayoutItem`.

Hier is een beschrijving van enkele van de belangrijkste items voor lay-out die aan een lay-out kunnen worden toegevoegd.

- 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

  ```
  map = QgsLayoutItemMap(layout)
  layout.addItem(map)
  ```

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

```
label = QgsLayoutItemLabel(layout)
label.setText("Hello world")
label.adjustSizeToText()
layout.addItem(label)
```

- legenda

```
legend = QgsLayoutItemLegend(layout)
legend.setLinkedMap(map) # map is an instance of QgsLayoutItemMap
layout.addItem(legend)
```

- schaalbalk

```
item = QgsLayoutItemScaleBar(layout)
item.setStyle('Numeric') # optionally modify the style
item.setLinkedMap(map) # map is an instance of QgsLayoutItemMap
item.applyDefaultSize()
layout.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))

polygonItem = QgsLayoutItemPolygon(polygon, layout)
layout.addItem(polygonItem)

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

symbol = QgsFillSymbol.createSimple(props)
polygonItem.setSymbol(symbol)
```

- tabel

Als een item eenmaal is toegevoegd aan de lay-out kan het worden verplaatst en de grootte worden gewijzigd:

```
item.attemptMove(QgsLayoutPoint(1.4, 1.8, QgsUnitTypes.LayoutCentimeters))
item.attemptResize(QgsLayoutSize(2.8, 2.2, QgsUnitTypes.LayoutCentimeters))
```

Standaard wordt een kader rondom elk item getekend. U kunt dat als volgt verwijderen:

```
# for a composer label
label.setFrameEnabled(False)
```

Naast het handmatig maken van items voor afdruklay-out, heeft QGIS ondersteuning voor sjablonen van afdruklay-out wat in essentie lay-outs zijn met al hun items, opgeslagen als een bestand .qpt (met syntaxis XML).

Als de lay-out eenmaal gereed is (de items van afdruklay-out zijn gemaakt en toegevoegd aan de lay-out), kunnen we doorgaan en een raster- en/of vector-uitvoer produceren.

9.3.1 Lay-out exporteren

De klasse `QgsLayoutExporter` moet worden gebruikt om een lay-out te exporteren.

```
base_path = os.path.join(QgsProject.instance().homePath()
pdf_path = os.path.join(base_path, "output.pdf")

exporter = QgsLayoutExporter(layout)
exporter.exportToPdf(pdf_path, QgsLayoutExporter.PdfExportSettings())
```

Gebruik `exportToImage()` in het geval dat u wilt exporteren naar een afbeelding in plaats van een bestand PDF.

9.3.2 Een afdrukatlas exporteren

Als u alle pagina's wilt exporteren van een lay-out die de optie Atlas heeft geconfigureerd en ingeschakeld, dient u de methode `atlas()` te gebruiken voor het exporteren (`QgsLayoutExporter`) met enkele kleine aanpassingen. In het volgende voorbeeld worden de pagina's geëxporteerd naar afbeeldingen PNG:

```
exporter.exportToImage(layout.atlas(), base_path, 'png', QgsLayoutExporter.
↪ImageExportSettings())
```

Onthoud dat de uitvoer zal worden opgeslagen in de map voor het basispad, met de expressie voor de bestandsnaam voor de uitvoer die werd geconfigureerd in Atlas.

De codesnippers op deze pagina hebben de volgende import nodig als u buiten de console van PyQGIS bent:

```
from qgis.core import (
   edit,
   QgsExpression,
   QgsExpressionContext,
   QgsFeature,
   QgsFeatureRequest,
   QgsField,
   QgsFields,
   QgsVectorLayer,
   QgsPointXY,
   QgsGeometry,
   QgsProject,
   QgsExpressionContextUtils
)
```

CHAPTER 10

Expressies, filteren en waarden berekenen

- *Parsen van expressies*
- *Evalueren van expressies*
  - *Basisexpressies*
  - *Expressies met objecten*
  - *Een laag filteren met expressies*
- *Fouten in expressies afhandelen*

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

Het volgende voorbeeld laat zien hoe te controleren of een bepaalde expressie juist kan worden geparsd:

```
exp = QgsExpression('1 + 1 = 2')
assert(not exp.hasParserError())

exp = QgsExpression('1 + 1 = ')
assert(exp.hasParserError())

assert(exp.parserErrorString() == '\nsyntax error, unexpected $end')
```

10.2 Evalueren van expressies

Expressies kunnen in verschillende contexten worden gebruikt, bijvoorbeeld om objecten te filteren of om nieuwe veldwaarden te berekenen. In alle gevallen moet de expressie worden geëvalueerd. Dat betekent dat de waarde ervan wordt berekent door de gespecificeerde stappen voor de berekening uit te voeren, wat eenvoudige rekenkundige of samengestelde expressies kunnen zijn.

10.2.1 Basisexpressies

Deze basis expressie evalueert tot 1, wat betekent dat hij waar is:

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

10.2.2 Expressies met objecten

Voor het evalueren van een expressie ten opzichte van een object dient een object `QgsExpressionContext` te worden gemaakt en doorgegeven aan de functie evaluate om de expressie toe te staan toegang te krijgen tot de veldwaarden van het object.

Het volgende voorbeeld laat zien hoe een object te maken met een veld "Column" en hoe dit object toe te voegen aan de context van de expressie.

```
fields = QgsFields()
field = QgsField('Column')
fields.append(field)
feature = QgsFeature()
feature.setFields(fields)
feature.setAttribute(0, 99)

exp = QgsExpression('"Column"')
context = QgsExpressionContext()
context.setFeature(feature)
assert(exp.evaluate(context) == 99)
```

Het volgende is een meer compleet voorbeeld van hoe expressies te gebruiken in de context van een vectorlaag, om nieuwe veldwaarden te kunnen berekenen:

```
from qgis.PyQt.QtCore import QVariant

# create a vector layer
vl = QgsVectorLayer("Point", "Companies", "memory")
pr = vl.dataProvider()
pr.addAttributes([QgsField("Name", QVariant.String),
                  QgsField("Employees",  QVariant.Int),
                  QgsField("Revenue", QVariant.Double),
                  QgsField("Rev. per employee", QVariant.Double),
                  QgsField("Sum", QVariant.Double),
                  QgsField("Fun", QVariant.Double)])
vl.updateFields()

# add data to the first three fields
my_data = [
    {'x': 0, 'y': 0, 'name': 'ABC', 'emp': 10, 'rev': 100.1},
    {'x': 1, 'y': 1, 'name': 'DEF', 'emp': 2, 'rev': 50.5},
    {'x': 5, 'y': 5, 'name': 'GHI', 'emp': 100, 'rev': 725.9}]

for rec in my_data:
    f = QgsFeature()
    pt = QgsPointXY(rec['x'], rec['y'])
    f.setGeometry(QgsGeometry.fromPointXY(pt))
    f.setAttributes([rec['name'], rec['emp'], rec['rev']])
    pr.addFeature(f)

vl.updateExtents()
QgsProject.instance().addMapLayer(vl)

# The first expression computes the revenue per employee.
# The second one computes the sum of all revenue values in the layer.
# The final third expression doesn't really make sense but illustrates
# the fact that we can use a wide range of expression functions, such
# as area and buffer in our expressions:
expression1 = QgsExpression('"Revenue"/"Employees"')
expression2 = QgsExpression('sum("Revenue")')
expression3 = QgsExpression('area(buffer($geometry,"Employees"))')

# QgsExpressionContextUtils.globalProjectLayerScopes() is a convenience
# function that adds the global, project, and layer scopes all at once.
# Alternatively, those scopes can also be added manually. In any case,
# it is important to always go from "most generic" to "most specific"
# scope, i.e. from global to project to layer
context = QgsExpressionContext()
context.appendScopes(QgsExpressionContextUtils.globalProjectLayerScopes(vl))

with edit(vl):
```

```
    for f in vl.getFeatures():
        context.setFeature(f)
        f['Rev. per employee'] = expression1.evaluate(context)
        f['Sum'] = expression2.evaluate(context)
        f['Fun'] = expression3.evaluate(context)
        vl.updateFeature(f)

print( f['Sum'])
```

10.2.3 Een laag filteren met expressies

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

```
layer = QgsVectorLayer("Point?field=Test:integer",
                           "addfeat", "memory")

layer.startEditing()

for i in range(10):
    feature = QgsFeature()
    feature.setAttributes([i])
    assert(layer.addFeature(feature))
layer.commitChanges()

expression = 'Test >= 3'
request = QgsFeatureRequest().setFilterExpression(expression)

matches = 0
for f in layer.getFeatures(request):
   matches += 1

assert(matches == 7)
```

10.3 Fouten in expressies afhandelen

Fouten die gerelateerd zijn aan expressies kunnen optreden tijdens het parsen of evalueren van de expressie:

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

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

De codesnippers op deze pagina hebben de volgende import nodig als u buiten de console van PyQGIS bent:

```
from qgis.core import (
  QgsProject,
  QgsSettings,
  QgsVectorLayer
)
```

CHAPTER 11

Instellingen lezen en opslaan

Waarschuwung: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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 kunnen onderscheid maken tussen de verschillende typen instellingen:

- **global settings** — they are bound to the user at a particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. Settings are handled using the `QgsSettings` class, through for example the `setValue()` and `value()` methods.

  Hier ziet u een voorbeeld van hoe deze methoden worden gebruikt.

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

def read():
  s = QgsSettings()
  mytext = s.value("myplugin/mytext", "default text")
  myint  = s.value("myplugin/myint", 123)
  myreal = s.value("myplugin/myreal", 2.71)
  nonexistent = s.value("myplugin/nonexistent", None)
  print(mytext)
  print(myint)
  print(myreal)
  print(nonexistent)
```

De tweede parameter van de methode `value()` is optioneel en specificeert de standaard waarde als er geen eerdere waarde is ingesteld voor de doorgegeven naam van de instelling.

- **projectinstellingen** — variëren tussen de verschillende projecten en daarom zijn ze gebonden aan een projectbestand. De kleur van de achtergrond van het kaartvenster of het doel coördinaten referentiesysteem (CRS) zijn daar voorbeelden van — een witte achtergrond en WGS84 zouden misschien geschikt zijn voor het ene project, terwijl een gele achtergrond en de projectie UTM beter geschikt zijn voor een ander.

  Een voorbeeld van het gebruik volgt nog.

```
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 (returns a tuple with the value, and a status boolean
# which communicates whether the value retrieved could be converted to
# its type, in these cases a string, an integer, a double and a boolean
# respectively)

mytext, type_conversion_ok = proj.readEntry("myplugin",
                                            "mytext",
                                            "default text")
myint, type_conversion_ok = proj.readNumEntry("myplugin",
                                              "myint",
                                              123)
mydouble, type_conversion_ok = proj.readDoubleEntry("myplugin",
                                                    "mydouble",
                                                    123)
mybool, type_conversion_ok = proj.readBoolEntry("myplugin",
                                                "mybool",
                                                123)
```

  Zoals u kunt zien wordt de methode `writeEntry()` gebruikt voor alle gegevenstypen, maar er bestaan verscheidene methoden om de waarde van de instelling terug in te lezen, en de corresponderende moet worden geselecteerd voor elk gegevenstype.

- **instellingen voor kaartlagen** — deze instellingen zijn gerelateerd aan een bepaalde instance van een kaartlaag in een project. Zij zijn *niet* verbonden met de onderliggende gegevensbron van een laag, dus als u twee instances voor kaartlagen maakt uit één shapefile, zullen zij de instellingen niet delen. De instellingen worden opgeslagen in het projectbestand, dus als de gebruiker het project opnieuw opent, zijn de aan de laag gerelateerde instellingen weer aanwezig. De waarde voor een opgegegevn instelling wordt opgehaald met behulp van de methode `customProperty()`, en kan worden ingesteld met behulp van de methode `setCustomProperty()`.

```
vlayer = QgsVectorLayer()
# save a value
vlayer.setCustomProperty("mytext", "hello world")

# read the value again (returning "default text" if not found)
mytext = vlayer.customProperty("mytext", "default text")
```

CHAPTER 12

Communiceren met de gebruiker

- *Berichten weergeven. De klasse QgsMessageBar*
- *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 Berichten weergeven. De klasse QgsMessageBar

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.core import Qgis
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that
↪", level=Qgis.Critical)
```

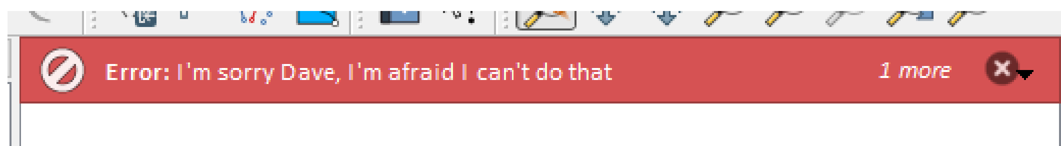


Figure 12.1: QGIS berichtenbalk

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

```
iface.messageBar().pushMessage("Ooops", "The plugin is not working as it should",␣
↪level=Qgis.Critical, duration=3)
```

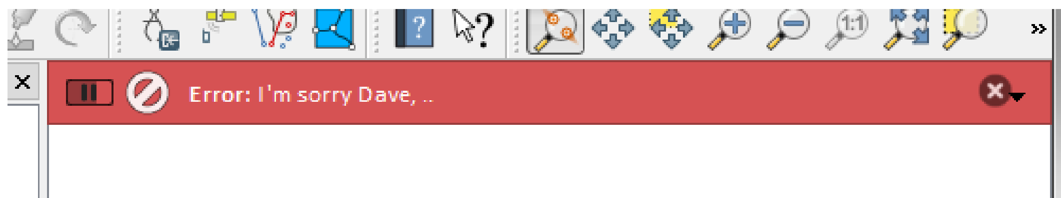


Figure 12.2: QGIS berichtenbalk met timer

Het voorbeeld hierboven geeft een foutenbalk weer, maar de parameter `level` kan worden gebruikt om waarschuwingen of informatie-berichten te maken, respectievelijk met behulp van de enumeratie `Qgis.MessageLevel`. U kunt tot maximaal 4 verschillende niveaus gebruiken.

0. Informatie
1. Waarschuwing
2. Kritisch
3. Succes

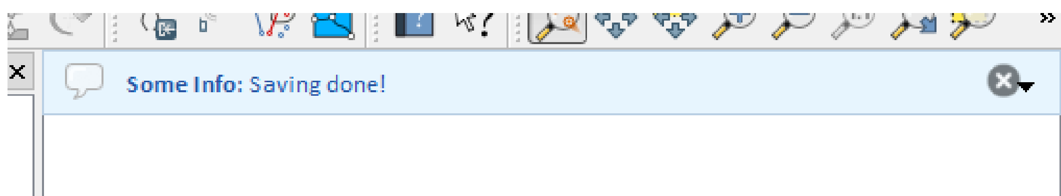


Figure 12.3: QGIS berichtenbalk (info)

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, Qgis.Warning)
```

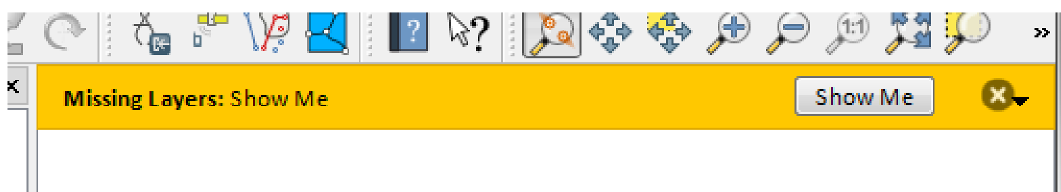


Figure 12.4: QGIS berichtenbalk met een knop

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=Qgis.Info)

myDlg = MyDialog()
myDlg.show()
```

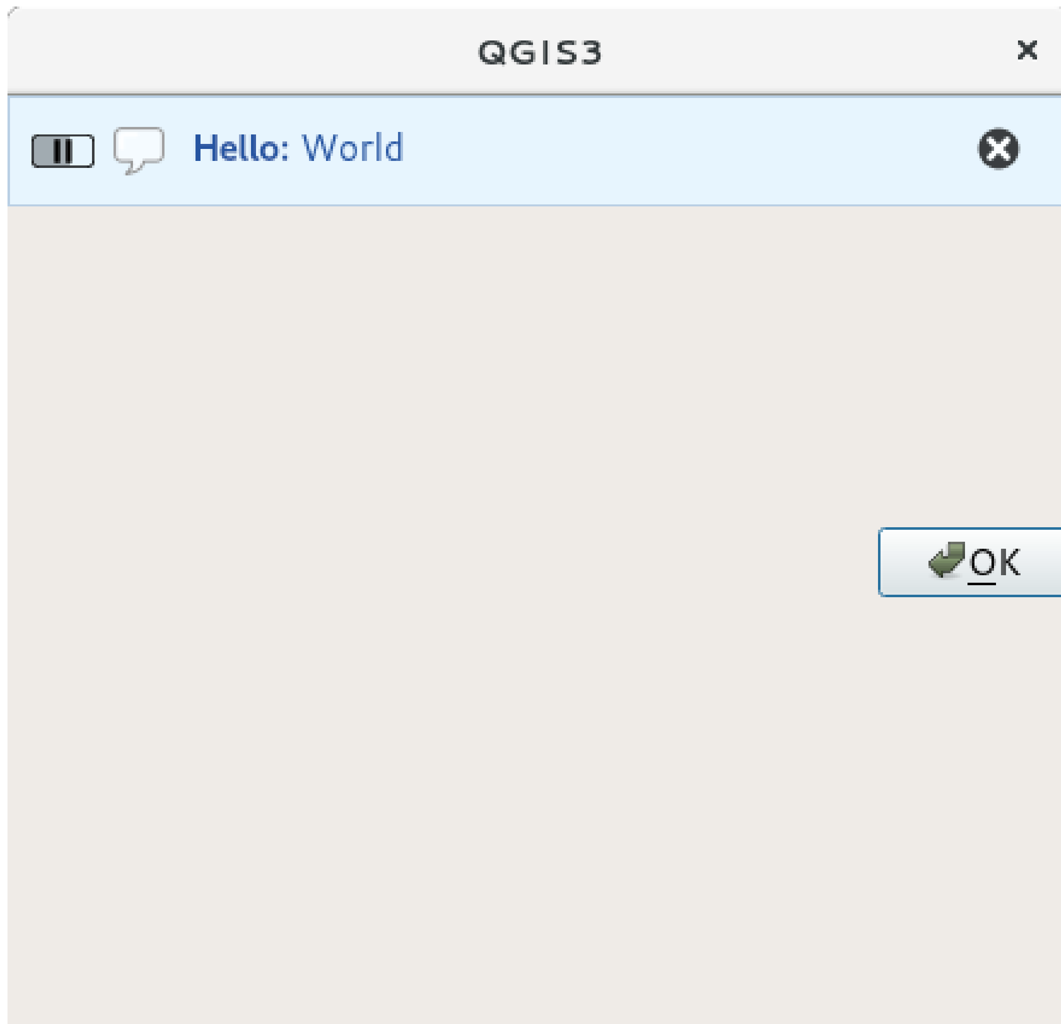


Figure 12.5: QGIS berichtenbalk in aangepast dialoogvenster

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 qgis.PyQt.QtWidgets import QProgressBar
from qgis.PyQt.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
```

```
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)

for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)

iface.messageBar().clearWidgets()
```

U kunt ook de ingebouwde statusbalk gebruiken om de voortgang weer te geven, zoals in het volgende voorbeeld:

```
vlayer = QgsProject.instance().mapLayersByName("countries")[0]

count = vlayer.featureCount()
features = vlayer.getFeatures()

for i, feature in enumerate(features):
    # do something time-consuming here
    print('') # printing should give enough time to present the progress

    percent = i / float(count) * 100
    # iface.mainWindow().statusBar().showMessage("Processed {} %".
↪format(int(percent)))
    iface.statusBarIface().showMessage("Processed {} %".format(int(percent)))

iface.statusBarIface().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
↪', level=Qgis.Info)
QgsMessageLog.logMessage("Your plugin code might have some problems", level=Qgis.
↪Warning)
QgsMessageLog.logMessage("Your plugin code has crashed!", level=Qgis.Critical)
```

Waarschuwing: het gebruiken van het argument in Python `print` is niet veilig om in enige code te gebruiken die multithreaded zou kunnen zijn. Dit omvat **functies voor expressies**, **renderers**, **symboollagen** en **algoritmen voor Processing** (naast andere). In deze gevallen zou u in plaats daarvan altijd thread-veilige klassen (`QgsLogger` of `QgsMessageLog`) moeten gebruiken.

Notitie: U kunt de uitvoer van `QgsMessageLog` zien in het log_message_panel

Notitie:

- `QgsLogger` is voor berichten voor debuggen / ontwikkelaars (d.i. als u vermoedt dat zij worden veroorzaakt door een beschadigde code)
- `QgsMessageLog` is voor berichten voor problemen die moeten worden onderzocht door systeembeheerders (bijv. om een systeembeheerder te helpen configuraties te herstellen)

CHAPTER 13

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*

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

13.1 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.

De meeste van de volgende snippers 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.

13.2 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 Een met een *Master Password* versleutelde Sqlite database `qgis-auth.db` waar *Configuratie voor authenticatie* worden opgeslagen. bijv gebruiker/wachtwoord, persoonlijke certificaten en sleutels, 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.

13.3 QgsAuthManager is het toegangspunt

De singleton `QgsAuthManager` is het toegangspunt om de in de versleutelde *Authentication DB* van QGIS opgeslagen gegevens te gebruiken, d.i. het bestand `qgis-auth.db` in de actieve map van het gebruikersprofiel.

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.

13.3.1 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" )
```

13.3.2 Vullen van authdb met een nieuw item Configuratie voor authenticatie

Elk opgeslagen persoonlijk gegeven is een instantie *Authentication Configuration* van de klasse `QgsAuthMethodConfig` waar toegang toe wordt verkregen met behulp van een unieke string zoals de volgende:

```
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 de basisklasse voor elke *Authentication Method*. Elke Authenticatiemethode stelt een configuratie hashmap in waar informatie voor authenticatie zal worden opgeslagen. Hieronder een handig snippet om persoonlijke gegevens voor het PKI-pad op te slaan voor een hypothetische gebruiker Alice:

```
authMgr = QgsAuthManager.instance()
# set alice PKI data
p_config = QgsAuthMethodConfig()
p_config.setName("alice")
p_config.setMethod("PKI-Paths")
p_config.setUri("https://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 Methods 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 Ontwikkeling 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

Een handige klasse om PKI-bundels in te pakken die zijn samengesteld uit de keten SslCert, SslKey en CA is de klasse `QgsPkiBundle`. Hieronder een snippet om wachtwoord beveiligd te verkrijgen:

```
# 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()
```

Bekijk de documentatie voor de klasse `QgsPkiBundle` om cert/key/CAs uit de bundel uit te nemen.

13.3.3 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" )
```

13.3.4 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 Configs* 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: Houdt er rekening mee dat niet alle gegevensproviders voor QGIS zijn geïntegreerd in de infrastructuur voor authenticatie. Elke authenticatiemethode, afgeleid van de basisklasse `QgsAuthMethod` ondersteunt een verschillende set van providers. Bijvoorbeeld: de methode `certIdentity ()` ondersteunt de volgende lijst met providers:

```
In [19]: authM = QgsAuthManager.instance()
In [20]: authM.authMethod("Identity-Cert").supportedDataProviders()
Out[20]: ['ows', 'wfs', 'wcs', 'wms', '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: De ontwikkelaar zou uitbreiding van `authcfg` moeten overlaten aan de `QgsAuthManager`, op deze manier zorgt hij er voor dat de uitbreiding niet te vroeg wordt gedaan.

Gewoonlijk wordt een tekenreeks als URI, gebouwd met behulp van de klasse `QgsDataSourceURI`, gebruikt om een gegevensbron voor QGIS op de volgende manier in te stellen:

```
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.

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

Vele plug-ins van derde partijen gebruiken httplib2 om verbindingen met HTTP te maken in plaats van ze te integreren met `QgsNetworkAccessManager` en de daaraan gerelateerde integratie voor de infrastructuur voor authenticatie. Om deze integratie te faciliteren is een hulpfunctie in Python gemaakt die is genaamd `NetworkAccessManager`. De code ervan is hier te vinden.

Deze hulpklasse kan worden gebruikt zoals in het volgende snippet:

```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
```

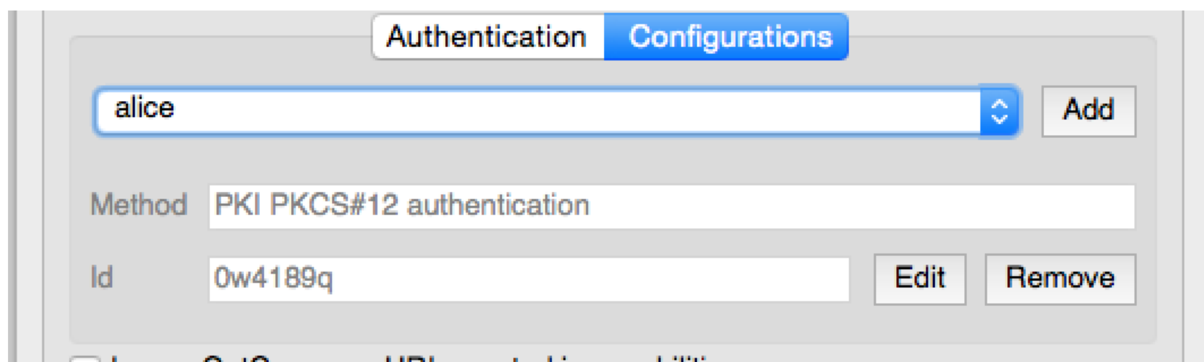
```
    response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
    # Handle exception
    pass
```

13.5 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.

13.5.1 GUI om persoonlijke gegevens te selecteren

Indien het noodzakelijk is een *Authentication Configuration* te selecteren uit een set die is opgeslagen in de *Authentication DB* is die beschikbaar in de klasse voor de GUI *QgsAuthConfigSelect <qgis.gui.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" )
```

Het bovenstaande voorbeeld is genomen uit de QGIS bron:source:*code <src/providers/postgres/qgspgnewconnection.cpp#L42>*. De tweede parameter van de constructor van de GUI verwijst naar het type gegevensprovider. De parameter wordt gebruikt om de compatibele *Authentication Method*en te beperken tot de gespecificeerde provider.

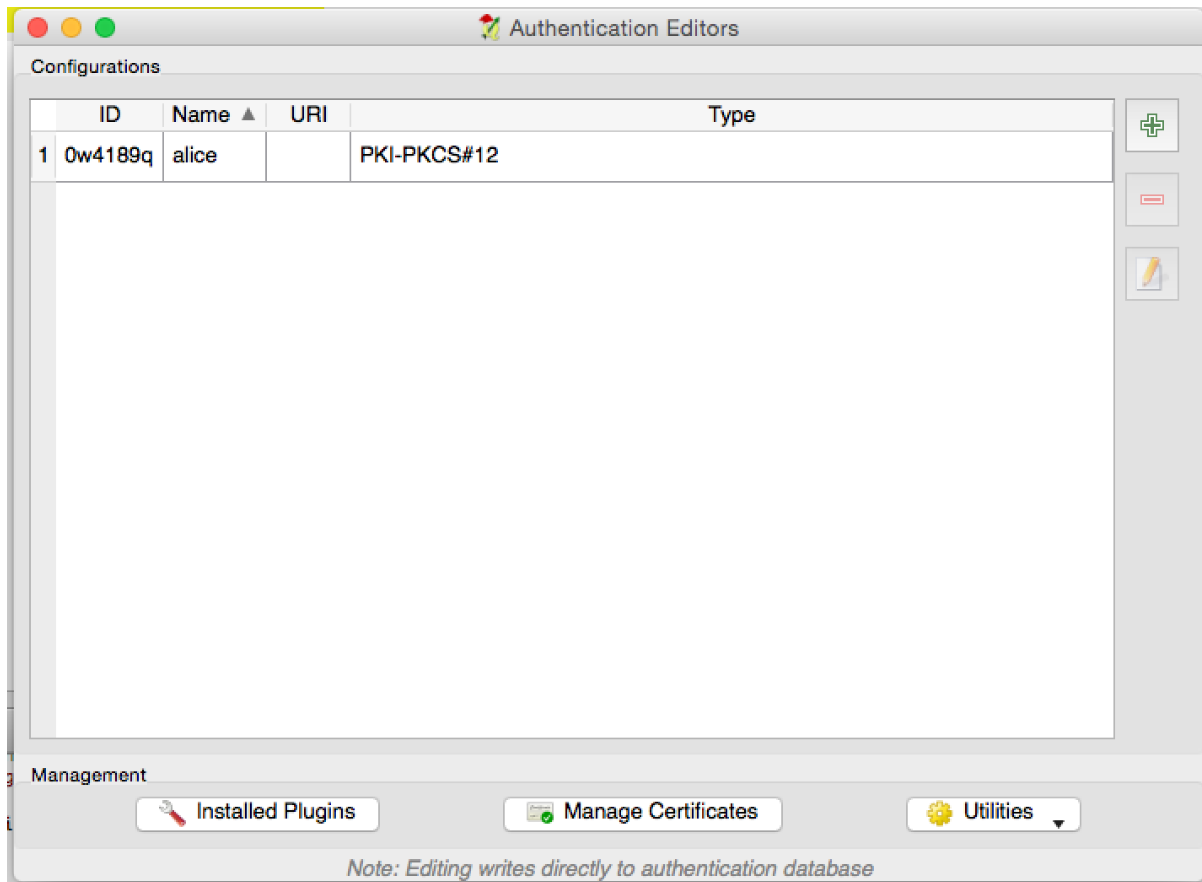
13.5.2 Bewerkers voor GUI authenticatie

De volledige GUI die wordt gebruikt voor het beheren van persoonlijke gegevens, autoriteiten en om toegang te verkrijgen tot de utilities voor authenticatie wordt beheerd door de klasse `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.

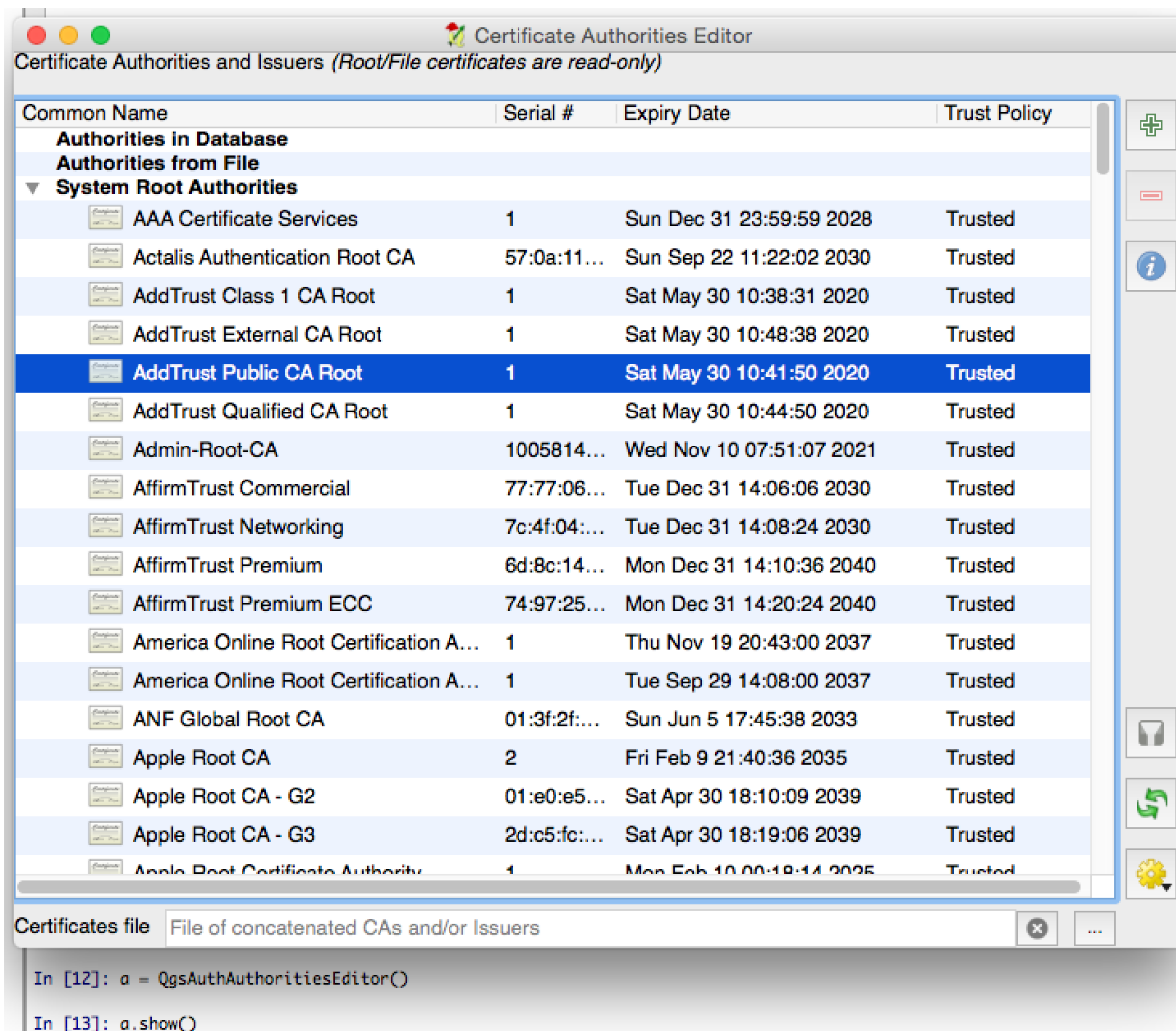


13.5.3 Bewerker voor GUI autoriteiten

Een GUI die alleen wordt gebruikt voor het beheren van autoriteiten wordt beheerd door de klasse *QgsAuthAuthoritiesEditor <qgis.gui.QgsAuthAuthoritiesEditor>*.

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()
```


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()

CHAPTER 14

Taken - veel werk op de achtergrond doen

14.1 Introductie

Verwerken op de achtergrond met behulp van threads is een manier om een reagerende gebruikersinterface te behouden wanneer veel verwerking wordt uitgevoerd. Taken kunnen worden gebruikt om threading te gebruiken in QGIS.

Een taak (`QgsTask`) is een container voor de code om op de achtergrond te wordne uitgevoerd, en de taakbeheerder (`QgsTaskManager`) wordt gebruikt om het uitvoeren van de taken te beheren. Deze klassen vereenvoudigen het verwken op de achtergrond in QGIS door mechanismen te verschaffen voor signaleren, voortgang rapporteren en toegang tot de status voor processen op de achtergrond. Taken kunnen worden gegroepeerd met behulp van subtaken.

Normaal gesproken wordt de globale taakbeheerder (te vinden met `QgsApplication.taskManager()`). Dit betekent dat uw taken niet de enige taken zijn die worden beheerd door de taakbeheerder.

Er zijn verschillende manieren om een taak voor QGIS te maken:

- Maak uw eigen taak door `QgsTask` uit te breiden

```
class SpecialisedTask(QgsTask):
```

- Maak een taak uit een functie.

```
QgsTask.fromFunction('heavy function', heavyFunction,
                     onfinished=workdone)
```

- Maak een taak uit een algoritme van Processing.

```
QgsProcessingAlgRunnerTask('native:buffer', params, context,
                           feedback)
```

Waarschuwing: Elke taak op de achtergrond (ongeacht hoe die is gemaakt) moet NOOIT op de GUI gebaseerde bewerkingen uitvoeren, zoals het maken van nieuwe widgets of interactief zijn met bestaande widgets. Toegang tot of aanpassen van widgets voor Qt moet alleen gebeuren vanuit de hoofdthread. Pogingen om ze te gebruiken vanuit threads voor de achtergrond zal leiden tot crashes.

Afhankelijkheden tussen taken kunnen worden beschreven met de functie addSubTask van QgsTask. Wanneer een afhankelijkheid wordt aangegeven,zal de taakbeheerder automatisch bepalen doe die afhankelijkheden zullen worden uitgevoerd. Waar mogelijk worden afhankelijkheden parallel uitgevoerd om ze zo snel als mogelijk voltooid te krijgen. Indien een taak waarvan een andere taak afhankelijk is wordt geannuleerd, zal de afhankelijke taak ook worden geannuleerd. Circulaire afhankelijkheden kunnen vastlopers mogelijk maken, wees dus voorzichtig.

Of een taak afhankelijk is van het feit of een laag beschikbaar is kan worden aangegeven met behulp van de functie setDependentLayers van QgsTask. Indien een laag, waarvan de taak afhankelijk is, niet beschikbaar is, wordt de taak geannuleerd.

Als de taak eenmaal is gemaakt kan die voor uitvoering in een schema worden geplaatst met behulp van de functie addTask van de taakbeheerder. Toevoegen van een taak aan de beheerder draagt automatisch het eigendom van die taak over aan de beheerder en de beheerder zal taken opschonen en verwijderen nadat zij zijn uitgevoerd. Het in schema zetten van de taken wordt beïnvloed door de prioriteit van de taak, die wordt ingesteld in addTask.

De status van taken kan worden gemonitord met behulp van signalen en functies van QgsTask en QgsTaskManager.

14.2 Voorbeelden

14.2.1 QgsTask uitbreiden

In dit voorbeeld breidt RandomIntegerSumTask QgsTask uit en zal 100 willekeurige integers tussen 0 en 500 genereren gedurende een gespecificeerde tijdperiode. Als het willekeurige getal 42 is zal de taak worden afgebroken en een uitzondering opgeworpen. Verscheidene instances van RandomIntegerSumTask (met subtaken) worden gemaakt en toegevoegd aan de taakbeheerder, wat twee typen afhankelijkheden demonstreert.

```
import random
from time import sleep

from qgis.core import (
    QgsApplication, QgsTask, QgsMessageLog,
    )

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""
    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None
    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
        Raising exceptions will crash QGIS, so we handle them
        internally and raise them in self.finished
        """
        QgsMessageLog.logMessage('Started task "{}"'.format(
                                     self.description()),
                                 MESSAGE_CATEGORY, Qgis.Info)
        wait_time = self.duration / 100
        for i in range(100):
            sleep(wait_time)
            # use setProgress to report progress
```

```
            self.setProgress(i)
            arandominteger = random.randint(0, 500)
            self.total += arandominteger
            self.iterations += 1
            # check isCanceled() to handle cancellation
            if self.isCanceled():
                return False
            # simulate exceptions to show how to abort task
            if arandominteger == 42:
                # DO NOT raise Exception('bad value!')
                # this would crash QGIS
                self.exception = Exception('bad value!')
                return False
        return True
    def finished(self, result):
        """
        This function is automatically called when the task has
        completed (successfully or not).
        You implement finished() to do whatever follow-up stuff
        should happen after the task is complete.
        finished is always called from the main thread, so it's safe
        to do GUI operations and raise Python exceptions here.
        result is the return value from self.run.
        """
        if result:
            QgsMessageLog.logMessage(
                'Task "{name}" completed\n' \
                'Total: {total} (with {iterations} '\
              'iterations)'.format(
                  name=self.description(),
                  total=self.total,
                  iterations=self.iterations),
              MESSAGE_CATEGORY, Qgis.Success)
        else:
            if self.exception is None:
                QgsMessageLog.logMessage(
                    'Task "{name}" not successful but without '\
                    'exception (probably the task was manually '\
                    'canceled by the user)'.format(
                        name=self.description()),
                    MESSAGE_CATEGORY, Qgis.Warning)
            else:
                QgsMessageLog.logMessage(
                    'Task "{name}" Exception: {exception}'.format(
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception
    def cancel(self):
        QgsMessageLog.logMessage(
            'Task "{name}" was canceled'.format(
                name=self.description()),
            MESSAGE_CATEGORY, Qgis.Info)
        super().cancel()


longtask = RandomIntegerSumTask('waste cpu long', 20)
shorttask = RandomIntegerSumTask('waste cpu short', 10)
minitask = RandomIntegerSumTask('waste cpu mini', 5)
shortsubtask = RandomIntegerSumTask('waste cpu subtask short', 5)
longsubtask = RandomIntegerSumTask('waste cpu subtask long', 10)
shortestsubtask = RandomIntegerSumTask('waste cpu subtask shortest', 4)
```

```
# Add a subtask (shortsubtask) to shorttask that must run after
# minitask and longtask has finished
shorttask.addSubTask(shortsubtask, [minitask, longtask])
# Add a subtask (longsubtask) to longtask that must be run
# before the parent task
longtask.addSubTask(longsubtask, [], QgsTask.ParentDependsOnSubTask)
# Add a subtask (shortestsubtask) to longtask
longtask.addSubTask(shortestsubtask)

QgsApplication.taskManager().addTask(longtask)
QgsApplication.taskManager().addTask(shorttask)
QgsApplication.taskManager().addTask(minitask)
```

14.2.2 Taak uit functie

Maak een taak uit een functie (in dit voorbeeld `doSomething`). De eerste parameter van de functie zal de klasse `QgsTask` voor de functie bevatten. Een belangrijke (benoemde) parameter is `on_finished`, die een functie specificeert die zal worden aangeroepen als de taak is voltooid. De functie `doSomething` in dit voorbeeld heeft een aanvullende benoemde parameter `wait_time`.

```
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

def doSomething(task, wait_time):
    """
    Raises an exception to abort the task.
    Returns a result if success.
    The result will be passed, together with the exception (None in
    the case of success), to the on_finished method.
    If there is an exception, there will be no result.
    """
    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
                             MESSAGE_CATEGORY, Qgis.Info)
    wait_time = wait_time / 100
    total = 0
    iterations = 0
    for i in range(100):
        sleep(wait_time)
        # use task.setProgress to report progress
        task.setProgress(i)
        arandominteger = random.randint(0, 500)
        total += arandominteger
        iterations += 1
        # check task.isCanceled() to handle cancellation
        if task.isCanceled():
            stopped(task)
            return None
        # raise an exception to abort the task
        if arandominteger == 42:
            raise Exception('bad value!')
    return {'total': total, 'iterations': iterations,
            'task': task.description()}

def stopped(task):
    QgsMessageLog.logMessage(
        'Task "{name}" was canceled'.format(
            name=task.description()),
        MESSAGE_CATEGORY, Qgis.Info)
```

```
def completed(exception, result=None):
    """This is called when doSomething is finished.
    Exception is not None if doSomething raises an exception.
    result is the return value of doSomething."""
    if exception is None:
        if result is None:
            QgsMessageLog.logMessage(
                'Completed with no exception and no result '\
                '(probably manually canceled by the user)',
                MESSAGE_CATEGORY, Qgis.Warning)
        else:
            QgsMessageLog.logMessage(
                'Task {name} completed\n'
                'Total: {total} ( with {iterations} '
                'iterations)'.format(
                    name=result['task'],
                    total=result['total'],
                    iterations=result['iterations']),
                MESSAGE_CATEGORY, Qgis.Info)
    else:
        QgsMessageLog.logMessage("Exception: {}".format(exception),
                                 MESSAGE_CATEGORY, Qgis.Critical)
        raise exception

# Creae a few tasks
task1 = QgsTask.fromFunction('Waste cpu 1', doSomething,
                             on_finished=completed, wait_time=4)
task2 = QgsTask.fromFunction('Waste cpu 2', doSomething,
                             on_finished=completed, wait_time=3)
QgsApplication.taskManager().addTask(task1)
QgsApplication.taskManager().addTask(task2)
```

14.2.3 Taak uit een algoritme voor Processing

Maak een taak die het algoritme qgis:randompointsinextent gebruikt om 50000 willekeurige punten te maken in een gespecificeerd bereik. Het resultaat wordt op een veilige manier aan het project toegevoegd.

```
from functools import partial
from qgis.core import (QgsTaskManager, QgsMessageLog,
                       QgsProcessingAlgRunnerTask, QgsApplication,
                       QgsProcessingContext, QgsProcessingFeedback,
                       QgsProject)

MESSAGE_CATEGORY = 'AlgRunnerTask'

def task_finished(context, successful, results):
    if not successful:
        QgsMessageLog.logMessage('Task finished unsucessfully',
                                 MESSAGE_CATEGORY, Qgis.Warning)
    output_layer = context.getMapLayer(results['OUTPUT'])
    # because getMapLayer doesn't transfer ownership, the layer will
    # be deleted when context goes out of scope and you'll get a
    # crash.
    # takeMapLayer transfers ownership so it's then safe to add it
    # to the project and give the project ownership.
    if output_layer and output_layer.isValid():
        QgsProject.instance().addMapLayer(
             context.takeResultLayer(output_layer.id()))

alg = QgsApplication.processingRegistry().algorithmById(
                                      'qgis:randompointsinextent')
```

```
context = QgsProcessingContext()
feedback = QgsProcessingFeedback()
params = {
    'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
    'MIN_DISTANCE': 0.0,
    'POINTS_NUMBER': 50000,
    'TARGET_CRS': 'EPSG:4326',
    'OUTPUT': 'memory:My random points'
}
task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
task.executed.connect(partial(task_finished, context))
QgsApplication.taskManager().addTask(task)
```

Zie ook: https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/.

CHAPTER 15

Python plug-ins ontwikkelen

15.1 Plug-ins voor Python structureren

- *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*
  - *Vertalen met behulp van Makefile*
  - *De plug-in laden*
- *Tips en trucs*
  - *Plugin Reloader*
  - *Toegang tot plug-ins*
  - *Logboekmeldingen*
  - *Uw plug-in delen*

Hier zijn enkele te volgen stappen voor het maken van een plug-in:

1. *Idee*: Weet u 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*: De noodzakelijke: Een beginpunt (`__init__.py`), vul de *Metadata van de plug-in* in `metadata.txt` in. Implementeer dan uw eigen ontwerp. Een hoofdgedeelte voor een plug-in in Python, bijv. `mainplugin.py`. Waarschijnlijk 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".

15.1.1 Een plug-in schrijven

Sinds de introductie van plug-ins voor Python in QGIS zijn een aantal plug-ins verschenen. Het team van QGIS onderhoudt een *Officiële Python plug-in opslagplaats*. U kunt hun bron gebruiken om meer te leren over programmeren met PyQGIS of uitzoeken of u een inspanning tot ontwikkeling dupliceert.

Plug-inbestanden

Hier is de mappenstructuur van onze voorbeeld-plug-in

```
PYTHON_PLUGINS_PATH/
  MyPlugin/
    __init__.py    --> *required*
    mainPlugin.py  --> *core code*
    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` = Bevat algemene informatie, versie, naam en enkele andere metadata, gebruikt door de website van de plug-in en infrastructuur van de plug-in.

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

Er is een plug-in voor QGIS, genaamd Plugin Builder 3 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 3.x.

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*

15.1.2 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.

Metadata van de plug-in

Als eerste moet Plug-ins beheren en installeren 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.

Notitie: Alle metadata moet in de 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	notatie, met punten, van de minimale versie van QGIS
qgisMaximumVersion	Nee	notatie, met punten, 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 notatie met punten
author	Ja	naam van de auteur
email	Ja	e-mail van de auteur, alleen weergegeven op de website voor ingelogde gebruikers, maar zichtbaar in Plug-ins beheren en installeren nadat de plug-in is geïnstalleerd
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=3.0
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=https://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=3.99
```

__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 object 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
```

mainPlugin.py

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

```
from qgis.PyQt.QtGui import *
from qgis.PyQt.QtWidgets import *

# initialize Qt resources from file resources.py
from . import resources

class TestPlugin:

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

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

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

  def renderTest(self, painter):
    # use painter for drawing to map canvas
```

```
    print("TestPlugin: renderTest called!")
```

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

In het voorbeeld hierboven wordt `addPluginToMenu` gebruikt. Dit zal de overeenkomstige actie voor het menu 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.mainWindow())
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    self.action.triggered.connect(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.

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 u nog slechts een bestand in Python te maken dat de bronnen zal bevatten. dat wordt gedaan met de opdracht **`pyrcc5`**

```
pyrcc5 -o resources.py resources.qrc
```

Notitie: In omgevingen van Windows zal het proberen uit te voeren van de opdracht **pyrcc5** 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\pyrcc5.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 Plug-ins beheren en installeren en een bericht in de console te zien wanneer het pictogram op de werkbalk of het menu-item 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 maken en de plug-in zal installeren in uw installatie van QGIS.

15.1.3 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.

15.1.4 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.

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 install qttools5-dev-tools
```

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 *voorbeeld plug-in*:

```
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 `pylupdate5`, 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 de 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 maken.

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

```
pylupdate5 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 typ:

```
lrelease your_plugin.ts
```

Nu zou u in de map `i18n` het/de bestand(en) `your_plugin_qm` moeten zien.

Vertalen met behulp van Makefile

Als u uw plug-in maakte met Plugin Builder kunt u als alternatief Makefile gebruiken om berichten uit te nemen uit code voor Python of dialoogvensters van Qt. Aan het begin van Makefile staat een variabele LOCALES:

```
LOCALES = en
```

Voeg de afkorting voor de taal toe aan deze variabele, bijvoorbeeld voor de Hongaarse taal:

```
LOCALES = en hu
```

Nu kunt u het bestand `hu.ts` (en het bestand `en.ts` ook) uit de bronnen maken of bijwerken met:

```
make transup
```

Hierna heeft u de bestanden `.ts` voor alle talen die zijn ingesteld in de variabele LOCALES bijgewerkt. Gebruik **Qt Linguist** om de berichten van het programma te vertalen. Als het vertalen voltooid is kunnen de bestanden `.qm` worden gemaakt met de transcompile:

```
make transcompile
```

U dient de bestanden `.ts` te distribueren met uw plug-in.

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.

15.1.5 Tips en trucs

Plugin Reloader

Tijdens het ontwikkelen van uw plug-in zult u die zeer regelmatig opnieuw moeten laden in QGIS om te testen. Dat gaat heel gemakkelijk met de plug-in Plugin Reloader. U vindt die als een experimentele plug-in in Plug-ins beheren en installeren.

Toegang tot plug-ins

U kunt toegang krijgen tot alle klassen van geïnstalleerde plug-ins vanuit QGIS met behulp van Python, wat handig kan zijn voor het debuggen.:

```
my_plugin = qgis.utils.plugins['My Plugin']
```

Logboekmeldingen

Plug-ins hebben hun eigen tab in het log_message_panel.

Uw plug-in delen

QGIS host honderden plug-ins in de opslagplaats voor plug-ins. Overweeg om de uwe te delen! Het zal de mogelijkheden van QGIS uitbreiden en mensen zullen in staat zijn te leren van uw code. Alle gehoste plug-ins kunnen in QGIS worden gevonden en geïnstalleerd met Plug-ins beheren en installeren.

Informatie en vereisten staan hier: plugins.qgis.org.

15.2 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.

15.2.1 Hoe een methode aan te roepen met een sneltoets

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

```
self.key_action = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.key_action, "Ctrl+I")  # action triggered␣
↪by Ctrl+I
self.iface.addPluginToMenu("&Test plugins", self.key_action)
self.key_action.triggered.connect(self.key_action_triggered)
```

Voeg aan `unload()` toe

```
self.iface.unregisterMainWindowAction(self.key_action)
```

De methode die wordt aangeroepen wanneer op CTRL+I wordt gedrukt

```
def key_action_triggered(self):
  QMessageBox.information(self.iface.mainWindow(),"Ok", "You pressed Ctrl+I")
```

15.2.2 Hoe te schakelen tussen lagen

Er is een API om toegang te verkrijgen tot lagen in de legenda. Hier is een voorbeeld dat schakelt 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.setItemVisibilityChecked(new_state)
```

15.2.3 Hoe toegang te krijgen tot de attributentabel van geselecteerde objecten

```
def change_value(value):
    """Change the value in the second column for all selected features.

    :param value: The new value.
    """
```

```
    layer = iface.activeLayer()
    if layer:
        count_selected = layer.selectedFeatureCount()
        if count_selected > 0:
            layer.startEditing()
            id_features = layer.selectedFeatureIds()
            for i in id_features:
                layer.changeAttributeValue(i, 1, value) # 1 being the second column
            layer.commitChanges()
        else:
            iface.messageBar().pushCritical("Error",
                "Please select at least one feature from current layer")
    else:
        iface.messageBar().pushCritical("Error", "Please select a layer")
```

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

```
changeValue(50)
```

15.3 Plug-in-lagen gebruiken

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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 om dat te implementeren.

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

15.3.1 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
```

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

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *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.

15.4.1 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
- 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
```

15.4.2 Debuggen met behulp van Eclipse en PyDev

Installatie

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

- Eclipse
- Aptana Studio 3 Plugin of PyDev
- QGIS 2.x

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 op de tab *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 van die te moeten sluiten en QGIS opnieuw op te moeten starten om hem opnieuw te laden.

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.

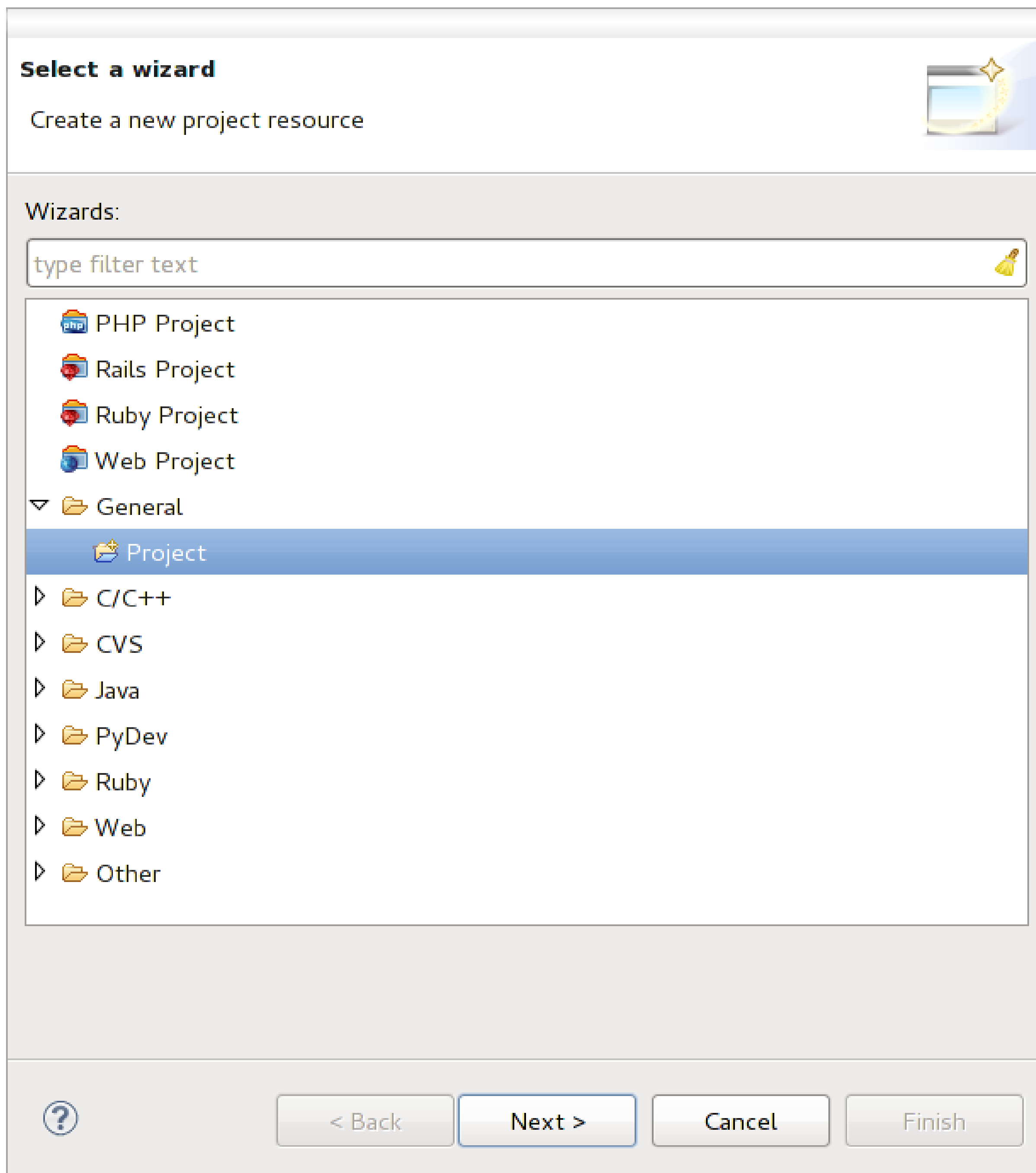


Figure 15.1: Eclipse-project

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.

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.

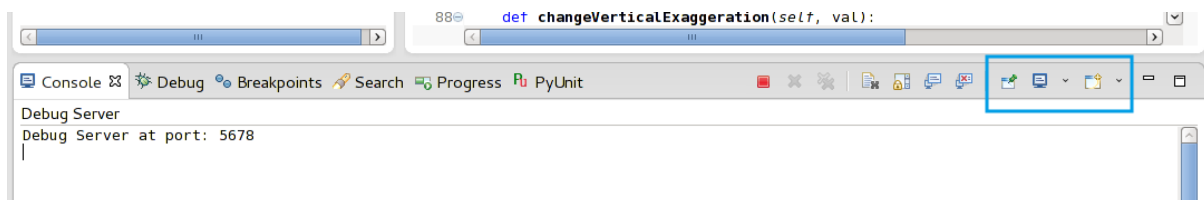


Figure 15.3: PyDev console voor debuggen

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,

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*.

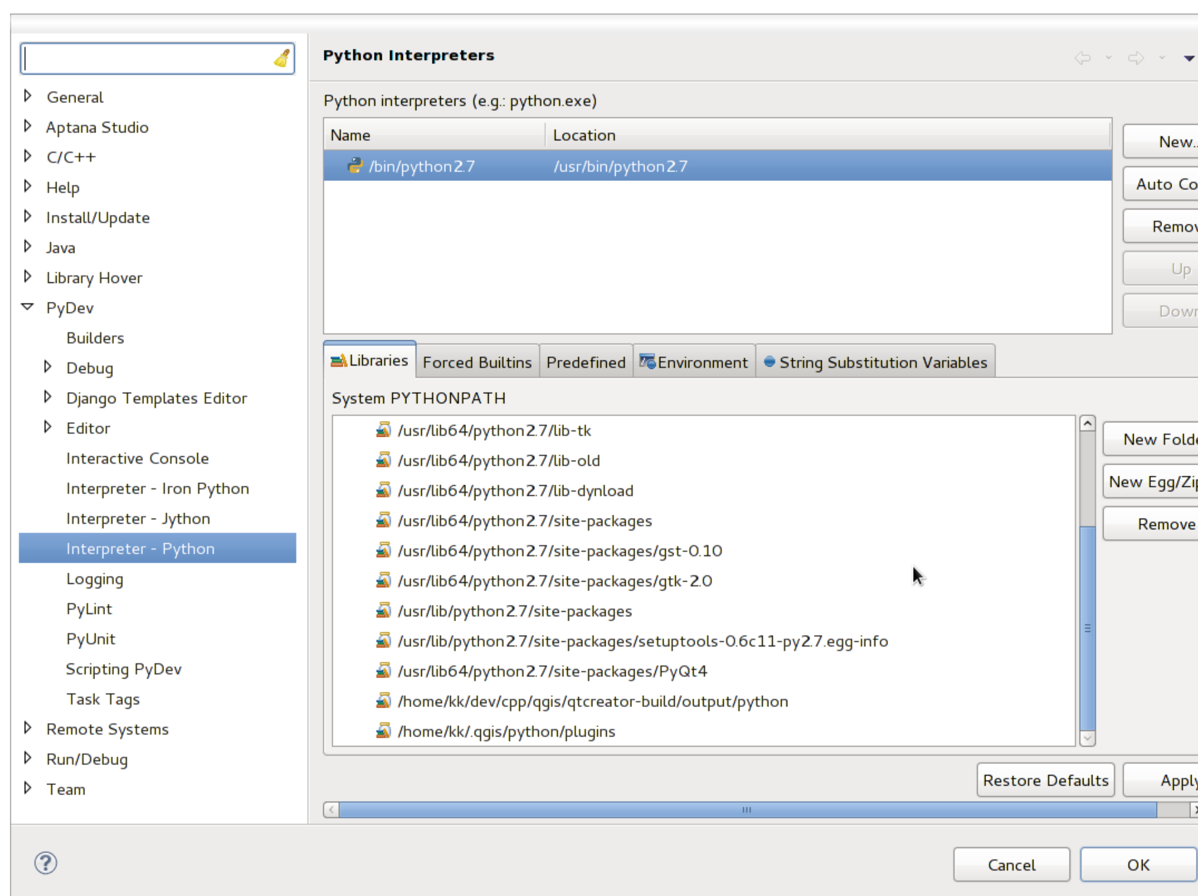


Figure 15.4: PyDev console voor debuggen

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 (het staat in `python/plugins` in de map gebruikersprofiel).

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 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.

15.4.3 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

15.5 Uw plug-in uitgeven

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *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.

Neem goede notie van de volgende suggesties:

15.5.1 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

15.5.2 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.

15.5.3 Officiële Python plug-in opslagplaats

U vindt de *officiële* Python plug-in opslagplaats op https://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

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.

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*.

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

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
```

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 `~/(UserProfile)/python/plugins` en deze paden:

- UNIX/Mac: `(qgis_prefix)/share/qgis/python/plugins`

- Windows: `(qgis_voorvoegsel)/python/plugins`

Voor definities van ~ en `(UserProfile)`, bekijk core_and_external_plugins.

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.

CHAPTER 16

Een plug-in voor Processing schrijven

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).

U zou, om deze algoritmen te kunnen distribueren, een nieuwe plug-in moeten maken die ze toevoegt aan de Toolbox van Processing. De plug-in zou een provider voor algoritmen moeten bevatten, die moet worden geregistreerd als de plug-in wordt geïnstantieerd.

U kunt de volgende stappen volgen, met behulp van de Plugin Builder, om vanaf nul een plug-in te maken die een provider voor algoritmen bevat:

- 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.

U dient nog enige code toe te voegen als u uw bestaande plug-in wilt toevoegen aan Processing.

In uw bestand `metadata.txt` dient u een variabele toe te voegen:

```
hasProcessingProvider=yes
```

In het bestand van Python waar uw plug-in wordt ingesteld met de methode `initGui`, dient u enkele regels als volgt aan te passen:

```
from qgis.core import QgsApplication
from .processing_provider.provider import Provider


class YourPluginName():

    def __init__(self):
        self.provider = None
```

```
    def initProcessing(self):
        self.provider = Provider()
        QgsApplication.processingRegistry().addProvider(self.provider)

    def initGui(self):
        self.initProcessing()

    def unload(self):
        QgsApplication.processingRegistry().removeProvider(self.provider)
```

U kunt een map `processing_provider` maken met daarin drie bestanden:

- `__init__.py` waar niets in staat. Dit is noodzakelijk om een geldig pakket voor Python te maken.
- `provider.py` dat de provider voor Processing zal maken en uw algoritmen zal laten zien.

```
from qgis.core import QgsProcessingProvider

from .example_processing_algorithm import ExampleProcessingAlgorithm


class Provider(QgsProcessingProvider):

    def loadAlgorithms(self, *args, **kwargs):
        self.addAlgorithm(ExampleProcessingAlgorithm())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())

    def id(self, *args, **kwargs):
        """The ID of your plugin, used for identifying the provider.

        This string should be a unique, short, character only string,
        eg "qgis" or "gdal". This string should not be localised.
        """
        return 'yourplugin'

    def name(self, *args, **kwargs):
        """The human friendly name of your plugin in Processing.

        This string should be as short as possible (e.g. "Lastools", not
        "Lastools version 1.0.1 64-bit") and localised.
        """
        return self.tr('Your plugin')

    def icon(self):
        """Should return a QIcon which is used for your provider inside
        the Processing toolbox.
        """
        return QgsProcessingProvider.icon(self)
```

- `example_processing_algorithm.py` dat het voorbeeldbestand voor het algoritme bevat. Kopieer/plak de inhoud van het sjabloonscript: https://github.com/qgis/QGIS/blob/release-3_4/python/plugins/processing/script/ScriptTemplate.py

Nu kunt u uw plug-in opnieuw laden in QGIS en u zou uw voorbeeldscript moeten zien in de Toolbox en Grafische modellen bouwen van Processing.

CHAPTER 17

Bibliotheek Netwerkanalyse

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *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.

17.1 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)

17.2 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.

Converteren van een vectorlaag naar de grafiek wordt gedaan met behulp van het Builder programmeringspatroon. Een grafiek wordt geconstrueerd met behulp van een zogenaamde Director. Er is nu nog slechts één Director: QgsLineVectorLayerDirector. De director stelt de basisinstellingen in die zullen worden gebruikt om een grafiek uit een lijn-vectorlaag te maken, gebruikt door de builder om de grafiek te maken. Momenteel, net zoals in het geval van de Director, bestaat er slechts één builder: `QgsGraphBuilder`, die objecten `QgsGraph` maakt. U wilt misschien uw eigen builders implementeren die een grafiek bouwen die compatibel is met bibliotheken zoals BGL of NetworkX.

Voor het berekenen van de eigenschappen van de rand wordt het programmeringspatroon strategy gebruikt. Momenteel is alleen beschikbaar QgsDistanceArcProperter strategie, dat rekening houdt met de lengte van de route. U kunt uw eigen strategie implementeren die alle noodzakelijke parameters zal gebruiken. De plug-in RoadGraph gebruikt bijvoorbeeld een strategie die de reistijd berekent met behulp van de lengte van de rand en een waarde voor de snelheid uit attributen.

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])
```

17.3 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

Gebruik, om de boom van het kortste pad te verkrijgen, de methoden `shortestTree` en `dijkstra` van de klasse `QgsGraphAnalyzer`. Aanbevolen wordt om de methode `dijkstra` te gebruiken omdat die sneller werkt en het geheugen meer efficiënt gebruikt.

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)
```

Hier is enige eenvoudige code om de boom van het kortste pad weer te geven met de grafiek die is gemaakt met de methode `shortestTree` (selecteer de lijnenlaag in het paneel *Lagen* en vervang de coördinaten door die van uzelf).

> **Waarschuwing:** Gebruik deze code alleen als voorbeeld, Het maakt heel veel objecten `QgsRubberBand` en zou zeer traag kunnen zijn voor grote gegevenssets.

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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 de methode `dijkstra`

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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())
```

17.3.1 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. Dan bouwen we met de methode `shortestTree` of `dijkstra` 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 lijnenlaag te selecteren in de inhoudsopgave en de coördinaten in de code te vervangen door die van uzelf) dat de methode `shortestTree` gebruikt

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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 voor de methode `dijkstra`

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

```
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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)
```

17.3.2 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 qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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 18

Python plug-ins voor QGIS server

Waarschuwing: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to https://qgis.org/pyqgis/master for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *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*

Plug-ins voor Python kunnen ook worden uitgevoerd op QGIS Server (bekijk label_qgisserver).

- Door de *server interface* (`QgsServerInterface`) te gebruiken kan een plug-in voor Python die wordt uitgevoerd op de server het gedrag van bestaande bronservices (**WMS**, **WFS** etc.) wijzigen.
- Met de *server filter interface* (`QgsServerFilter`) kunt u de parameters voor de invoer wijzigen, de gegenereerde uitvoer wijzigen of zelfs door nieuwe services te verschaffen.
- Met de *access control interface* (`QgsAccessControlFilter`) kunt u enkele toegangsbeperkingen per verzoeken toepassen.

18.1 Server Filter Plugins architecture

Server plug-ins voor Python worden geladen als eenmaal de toepassing FCGI is gestart. Zij registreren één of meerdere `QgsServerFilter` (op dit punt is het misschien nuttig om even snel te kijken naar server plugins API docs). Elk filter zou tenminste één van drie terugkoppelingen moeten implementeren:

- `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`
  - filters voor plug-ins aanroepen `requestReady`
  - **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
    - filters voor plug-ins aanroepen `responseComplete`
  - filters voor plug-ins aanroepen `sendResponse`
  - afhandeling van het verzoek voert het antwoord uit

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

18.1.1 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).

18.1.2 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` normaal 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 :meth:‘sendResponse <qgis.server.QgsServerFilter.sendResponse> de enige geldige optie in het geval van stromende services.

18.1.3 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).

18.2 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.

18.3 Een plug-in voor de server schrijven

Een plug-in voor de server is een standaard plug-in in Python voor QGIS Python zoals beschreven in *Python plug-ins ontwikkelen*, dat eenvoudigweg een aanvullende (of alternatieve) interface verschaft: een typische plug-in voor QGIS Desktop heeft toegang tot de toepassing QGIS via de instantie `QgisInterface`, een plug-in voor de server heeft ook toegang tot een `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
```

De hier besproken voorbeeldplug-in (met nog veel meer voorbeeldfilters) is beschikbaar op github: QGIS HelloServer Example Plugin. U kunt nog meer voorbeelden vinden op https://github.com/elpaso/qgis3-server-vagrant/tree/master/resources/web/plugins of door door de QGIS opslagplaats voor plug-ins te bladeren.

18.3.1 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*
```

18.3.2 __init__.py

Dit bestand wordt vereist door het systeem voor importeren van Python. Ook vereist QGIS Server dat dit bestand een functie `classFactory()` bevat, die wordt aangeroepen als de plug-in wordt geladen in QGIS Server. Het ontvangt een verwijzing naar de instantie van `QgsServerInterface` en moet een instantie teruggeven van de klasse van uw plug-in. Dit is hoe de voorbeeldplug-in `__init__.py` er uit ziet:

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

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

18.3.3 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 QgsServerFilter.

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` stelt een prioriteit in 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:

18.3.4 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', QgsMessageLog.INFO)
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete",
↪'plugin', QgsMessageLog.CRITICAL)
```

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]␣
↪HelloServerServer - loading filter ParamsFilter
 src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0]␣
↪Server plugin HelloServer loaded!
```

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0]␣
↪Server python plugins loaded
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is:␣
↪SERVICE=HELLO&request=GetOutput
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms]␣
↪inserting pair SERVICE // HELLO into the parameter map
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms]␣
↪inserting pair REQUEST // GetOutput into the parameter map
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter␣
↪plugin default requestReady called
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪HelloFilter.requestReady
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default␣
↪configuration file path: /home/xxx/apps/bin/admin.sld
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking␣
↪byte array is ok to set...
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array␣
↪looks good, setting response...
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪HelloFilter.responseComplete
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪SUCCESS - ParamsFilter.responseComplete
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪RemoteConsoleFilter.responseComplete
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.sendResponse
```

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).

18.3.5 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:

```python
import os

from qgis.server import *
from qgis.core import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.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 {}".format(request.infoFormat()), 'plugin', QgsMessageLog.INFO)
            # 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.

18.4 Plug-in Access control

18.4.1 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*
```

18.4.2 __init__.py

Dit bestand wordt vereist door het systeem voor importeren van Python. Net als voor alle plug-ins voor QGIS Server bevat dit bestand een functie `classFactory()` bevat, die wordt aangeroepen als de plug-in wordt geladen in QGIS Server bij het opstarten. Het ontvangt een verwijzing naar een instantie van `QgsServerInterface` en moet een instantie teruggeven van de klasse van uw plug-in. Dit is hoe de voorbeeldplug-in `__init__.py` er uit ziet:

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

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

18.4.3 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.

18.4.4 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”.

18.4.5 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”.

18.4.6 layerPermissions

Toegang beperken tot de laag.

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

- `canRead` om het te zien in de `GetCapabilities` en rechten voor lezen hebben.
- `canInsert` om een nieuw object te kunnen invoegen.
- `canUpdate` om een object te kunnen bijwerken.
- `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.

18.4.7 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.

18.4.8 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.

18.4.9 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.

CHAPTER 19

Cheatsheet voor PyQGIS

19.1 Gebruikersinterface

Uiterlijk wijzigingen

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
qss_file = open(r"/path/to/style/file.qss").read()
app.setStyleSheet(qss_file)
```

Pictogram en titel wijzigen

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon(r"/path/to/logo/file.png")
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("My QGIS")
```

19.2 Instellingen

Lijst QSettings ophalen

```
from qgis.PyQt.QtCore import QSettings

qs = QSettings()

for k in sorted(qs.allKeys()):
    print (k)
```

19.3 Werkbalken

Werkbalk verwijderen

```
from qgis.utils import iface

toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)

# and add again
parent.addToolBar(toolbar)
```

Acties van werkbalk verwijderen

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

19.4 Menu's

Menu verwijderen

```
from qgis.utils import iface

# for example Help Menu
menu = iface.helpMenu()
menubar = menu.parentWidget()
menubar.removeAction(menu.menuAction())

# and add again
menubar.addAction(menu.menuAction())
```

19.5 Kaartvenster

Toegang tot kaartvenster

```
from qgis.utils import iface

canvas = iface.mapCanvas()
```

Kleur kaartvenster wijzigen

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

Interval voor bijwerken kaart

```
from qgis.PyQt.QtCore import QSettings
# Set milliseconds (150 milliseconds)
QSettings().setValue("/qgis/map_update_interval", 150)
```

19.6 Lagen

Vectorlaag toevoegen

```
from qgis.utils import iface

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

Actieve laag ophalen

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

Alle lagen vermelden

```
from qgis.core import QgsProject

QgsProject.instance().mapLayers().values()
```

Namen van lagen ophalen

```
layers_names = []
for layer in QgsProject.instance().mapLayers().values():
    layers_names.append(layer.name())

print("layers TOC = {}".format(layers_names))
```

Anders

```
layers_names = [layer.name() for layer in QgsProject.instance().mapLayers().
↪values()]
print("layers TOC = {}".format(layers_names))
```

Laag zoeken op naam

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
print(layer.name())
```

Actieve laag instellen

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
iface.setActiveLayer(layer)
```

Laag vernieuwen met interval

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)
# Enable auto refresh
layer.setAutoRefreshEnabled(True)
```

Methoden weergeven

```
dir(layer)
```

Nieuw object met objectformulier toevoegen

```
from qgis.core import QgsFeature, QgsGeometry

feat = QgsFeature()
geom = QgsGeometry()
feat.setGeometry(geom)
feat.setFields(layer.fields())

iface.openFeatureForm(layer, feat, False)
```

Nieuw object zonder objectformulier toevoegen

```
from qgis.core import QgsPointXY

pr = layer.dataProvider()
feat = QgsFeature()
feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
pr.addFeatures([feat])
```

Objecten ophalen

```
for f in layer.getFeatures():
    print (f)
```

Geselecteerde objecten ophalen

```
for f in layer.selectedFeatures():
    print (f)
```

ID's van geselecteerde objecten ophalen

```
selected_ids = layer.selectedFeatureIds()
print(selected_ids)
```

Geheugenlaag uit ID's van geselecteerde objecten maken

```
from qgis.core import QgsFeatureRequest

memory_layer = layer.materialize(QgsFeatureRequest().setFilterFids(layer.
↪selectedFeatureIds()))
QgsProject.instance().addMapLayer(memory_layer)
```

Geometrie ophalen

```
# Point layer
for f in layer.getFeatures():
    geom = f.geometry()
    print ('%f, %f' % (geom.asPoint().y(), geom.asPoint().x()))
```

Geometrie verplaatsen

```
geom.translate(100, 100)
poly.setGeometry(geom)
```

CRS instellen

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.
↪EpsgCrsId))
```

CRS weergeven

```
from qgis.core import QgsProject

for layer in QgsProject.instance().mapLayers().values():
    crs = layer.crs().authid()
    layer.setName('{} ({})'.format(layer.name(), crs))
```

Een veldkolom verbergen

```
from qgis.core import QgsEditorWidgetSetup

def fieldVisibility (layer,fname):
    setup = QgsEditorWidgetSetup('Hidden', {})
    for i, column in enumerate(layer.fields()):
        if column.name()==fname:
            layer.setEditorWidgetSetup(idx, setup)
            break
        else:
            continue
```

Laag uit WKT

```
from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject

layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.0934.89,-88.39 30.34,-89.57␣
↪30.18,-89.73 31,-91.63 30.99,-90.8732.37,-91.23 33.44,-90.93 34.23,-90.30 34.99,-
↪88.82 34.99))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])
```

Alle lagen uit GeoPackage laden

```
from qgis.core import QgsVectorLayer, QgsProject

fileName = "/path/to/gpkg/file.gpkg"
layer = QgsVectorLayer(fileName,"test","ogr")
subLayers =layer.dataProvider().subLayers()

for subLayer in subLayers:
    name = subLayer.split('!!::!!')[1]
    uri = "%s|layername=%s" % (fileName, name,)
    # Create layer
    sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
    # Add layer to map
    QgsProject.instance().addMapLayer(sub_vlayer)
```

Tegellaag (XYZ-laag) laden

```
from qgis.core import QgsRasterLayer, QgsProject

def loadXYZ(url, name):
    rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
    QgsProject.instance().addMapLayer(rasterLyr)

urlWithParams = 'type=xyz&url=https://a.tile.openstreetmap.org/%7Bz%7D/%7Bx%7D/%7By
↪%7D.png&zmax=19&zmin=0&crs=EPSG3857'
loadXYZ(urlWithParams, 'OpenStreetMap')
```

Alle lagen verwijderen

```
QgsProject.instance().removeAllMapLayers()
```

Alles verwijderen

```
QgsProject.instance().clear()
```

19.7 Inhoud

Toegang tot geselecteerde lagen

```
from qgis.utils import iface

iface.mapCanvas().layers()
```

Contextmenu verwijderen

```
ltv = iface.layerTreeView()
mp = ltv.menuProvider()
ltv.setMenuProvider(None)
# Restore
ltv.setMenuProvider(mp)
```

19.8 Uitgebreide inhoud

Bronknoop

```
from qgis.core import QgsProject

root = QgsProject.instance().layerTreeRoot()
print (root)
print (root.children())
```

Toegang tot de eerste kindknoop

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer, QgsLayerTree

child0 = root.children()[0]
print (child0.name())
print (type(child0))
print (isinstance(child0, QgsLayerTreeLayer))
print (isinstance(child0.parent(), QgsLayerTree))
```

Groepen en knopen zoeken

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer

def get_group_layers(group):
   print('- group: ' + group.name())
   for child in group.children():
      if isinstance(child, QgsLayerTreeGroup):
         # Recursive call to get nested groups
         get_group_layers(child)
      else:
         print('  - layer: ' + child.name())


root = QgsProject.instance().layerTreeRoot()
```

```
for child in root.children():
    if isinstance(child, QgsLayerTreeGroup):
        get_group_layers(child)
    elif isinstance(child, QgsLayerTreeLayer):
        print ('- layer: ' + child.name())
```

Groep op naam zoeken

```
print (root.findGroup("My Group"))
```

Laag toevoegen

```
from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
```

Groep toevoegen

```
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
```

Laag verwijderen

```
root.removeLayer(layer1)
```

Groep verwijderen

```
root.removeChildNode(node_group2)
```

Knoop verplaatsen

```
cloned_group1 = node_group.clone()
root.insertChildNode(0, cloned_group1)
root.removeChildNode(node_group)
```

Knoop hernoemen

```
cloned_group1.setName("Group X")
node_layer1.setName("Layer X")
```

Zichtbaarheid wijzigen

```
print (cloned_group1.isVisible())
cloned_group1.setItemVisibilityChecked(False)
node_layer1.setItemVisibilityChecked(False)
```

Knoop uitbreiden

```
print (cloned_group1.isExpanded())
cloned_group1.setExpanded(False)
```

Truc verborgen knoop

```
from qgis.core import QgsProject

model = iface.layerTreeView().layerTreeModel()
ltv = iface.layerTreeView()
root = QgsProject.instance().layerTreeRoot()
```

```
layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
node=root.findLayer( layer.id())

index = model.node2index( node )
ltv.setRowHidden( index.row(), index.parent(), True )
node.setCustomProperty( 'nodeHidden', 'true')
ltv.setCurrentIndex(model.node2index(root))
```

Signalen voor knopen

```
def onWillAddChildren(node, indexFrom, indexTo):
    print ("WILL ADD", node, indexFrom, indexTo)

def onAddedChildren(node, indexFrom, indexTo):
    print ("ADDED", node, indexFrom, indexTo)

root.willAddChildren.connect(onWillAddChildren)
root.addedChildren.connect(onAddedChildren)
```

Nieuwe inhoudsopgave maken (TOC)

```
from qgis.core import QgsProject, QgsLayerTreeModel
from qgis.gui import QgsLayerTreeView

root = QgsProject.instance().layerTreeRoot()
model = QgsLayerTreeModel(root)
view = QgsLayerTreeView()
view.setModel(model)
view.show()
```

19.9 Algoritmes voor Processing

Lijst algoritmes ophalen

```
from qgis.core import QgsApplication

for alg in QgsApplication.processingRegistry().algorithms():
    print("{}:{} --> {}".format(alg.provider().name(), alg.name(), alg.
↪displayName()))
```

Anders

```
def alglist():
    s = ''
    for i in QgsApplication.processingRegistry().algorithms():
        l = i.displayName().ljust(50, "-")
        r = i.id()
        s += '{}--->{}\n'.format(l, r)
    print(s)
```

Help voor algoritmes ophalen

Willekeurige selectie

```
import processing

processing.algorithmHelp("qgis:randomselection")
```

Het algoritme uitvoeren

Voor dit voorbeeld wordt het resultaat opgeslagen in een tijdelijke geheugenlaag die wordt toegevoegd aan het project.

```
import processing
result = processing.run("native:buffer", {'INPUT': layer, 'OUTPUT': 'memory:'})
QgsProject.instance().addMapLayer(result['OUTPUT'])
```

Hoeveel algoritmes zijn er?

```
from qgis.core import QgsApplication

len(QgsApplication.processingRegistry().algorithms())
```

Hoeveel providers zijn er?

```
from qgis.core import QgsApplication

len(QgsApplication.processingRegistry().providers())
```

Hoeveel expressies zijn er?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

19.10 Decoraties

CopyRight

```
from qgis.PyQt.Qt import QTextDocument
from qgis.PyQt.QtGui import QFont

mQFont = "Sans Serif"
mQFontsize = 9
mLabelQString = "© QGIS 2019"
mMarginHorizontal = 0
mMarginVertical = 0
mLabelQColor = "#FF0000"

INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
case = 2

def add_copyright(p, text, xOffset, yOffset):
    p.translate( xOffset , yOffset  )
    text.drawContents(p)
    p.setWorldTransform( p.worldTransform() )

def _on_render_complete(p):
    deviceHeight = p.device().height() # Get paint device height on which this␣
↪painter is currently painting
    deviceWidth  = p.device().width() # Get paint device width on which this␣
↪painter is currently painting
    # Create new container for structured rich text
    text = QTextDocument()
    font = QFont()
    font.setFamily(mQFont)
    font.setPointSize(int(mQFontsize))
    text.setDefaultFont(font)
    style = "<style type=\"text/css\"> p {color: " + mLabelQColor + "}</style>"
    text.setHtml( style + "<p>" + mLabelQString + "</p>" )
```

```
    # Text Size
    size = text.size()

    # RenderMillimeters
    pixelsInchX  = p.device().logicalDpiX()
    pixelsInchY  = p.device().logicalDpiY()
    xOffset  = pixelsInchX  * INCHES_TO_MM * int(mMarginHorizontal)
    yOffset  = pixelsInchY  * INCHES_TO_MM * int(mMarginVertical)

    # Calculate positions
    if case == 0:
        # Top Left
        add_copyright(p, text, xOffset, yOffset)

    elif case == 1:
        # Bottom Left
        yOffset = deviceHeight - yOffset - size.height()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 2:
        # Top Right
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 3:
        # Bottom Right
        yOffset  = deviceHeight - yOffset - size.height()
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 4:
        # Top Center
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

    else:
        # Bottom Center
        yOffset = deviceHeight - yOffset - size.height()
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

# Emitted when the canvas has rendered
iface.mapCanvas().renderComplete.connect(_on_render_complete)
# Repaint the canvas map
iface.mapCanvas().refresh()
```

19.11 Bronnen

- QGIS Python (PyQGIS) API
- QGIS C++ API
- StackOverFlow QGIS questions
- Script van Klas Karlsson
- Boundless lib-qgis-common repository