PyQGIS 3.40 developer cookbook

QGIS Project

apr 03, 2025

Inhoudsopgave

HOOFDSTUK 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.

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
- Algoritmes voor Processing maken
- Functies voor expressies in QGIS maken
- Aangepaste toepassingen maken, gebaseerd op de API van QGIS

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

Er is een verwijzing complete C++ 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 andere goede bron voor het leren hoe algemene taken uit te voeren is om bestaande plug-ins te downloaden vanaf de opslagplaats voor plug-ins 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*:

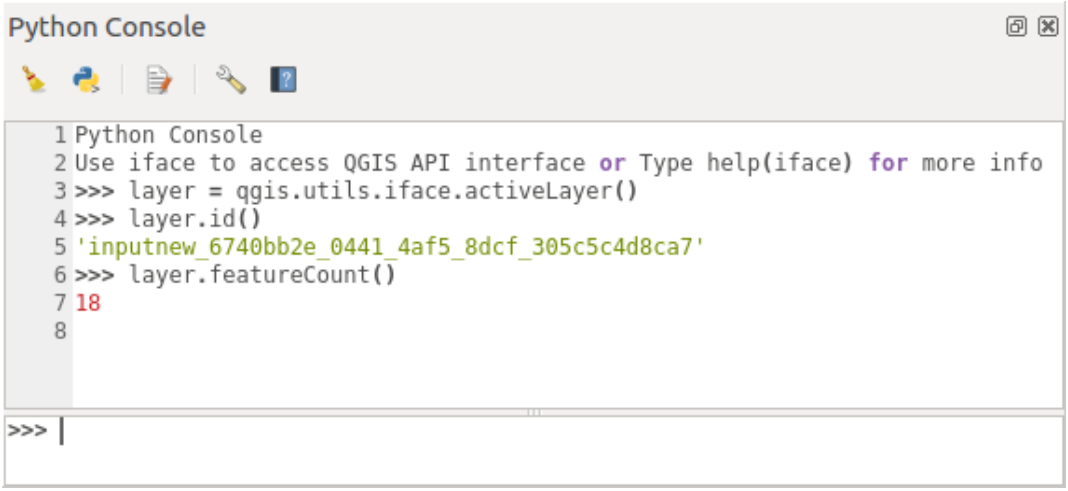


Fig. 1.1: QGIS Python-console

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 *QGIS Server en Python* voor meer details.

1.2.1 Processing plug-ins

Processing plug-ins kunnen worden gebruikt om gegevens te verwerken. Zij zijn eenvoudiger te ontwikkelen, meer specifiek en minder zwaar dan Python plug-ins. *Een plug-in voor Processing schrijven* legt uit wanneer het gebruiken van algoritmes van Processing toepasselijk is en hoe ze te ontwikkelen.

1.3 Python-code uitvoeren wanneer QGIS start

Er zijn verschillende methoden om code voor Python 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
3. Specificeren van een opstart-script met de parameter `--code init_qgis.py`.

1.3.1 Het bestand `startup.py`

Elke keer als QGIS start, wordt in de Python-thuismap van de gebruiker en een lijst met systeempaden gezocht naar een bestand genaamd `startup.py`. Als dat bestand bestaat, wordt het uitgevoerd door de ingebedde Python-interpreter.

Het pad in de thuismap van de gebruiker is meestal te vinden onder:

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

De standaard systeempaden zijn afhankelijk van het besturingssysteem. Open, om de paden te zoeken die voor u zullen werken, de console voor Python en voer `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` uit om de lijst met de standaard mappen te zien.

Het script `startup.py` wordt onmiddellijk uitgevoerd bij het initialiseren van Python in QGIS, heel vroeg bij het starten van de toepassing.

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.

1.3.3 De parameter `--code`

U kunt aangepaste code verschaffen om te worden uitgevoerd als parameter voor het opstarten van QGIS. Maak, om dat te doen, een bestand voor Python, bijvoorbeeld `qgis_init.py`, om QGIS uit te voeren en te starten vanaf de opdrachtregel met `qgis --code qgis_init.py`.

Code die is verschaft via `--code` wordt later in de fase van configuratie van QGIS uitgevoerd, nadat de componenten voor de toepassing zijn geladen.

1.3.4 Aanvullende argumenten voor Python

U kunt het argument `--py-args` gebruiken om aanvullende argumenten te verschaffen voor uw script `--code` of voor andere code voor Python die wordt uitgevoerd. Elk argument dat komt na `--py-args` en vóór een argument `--` (indien aanwezig) zal worden doorgegeven aan Python, maar worden genegeerd door de toepassing QGIS zelf.

In het volgende voorbeeld zal `myfile.tif` beschikbaar zijn via `sys.argv` in Python, maar zal niet worden geladen door QGIS. Waar `otherfile.tif` zal worden geladen door QGIS, maar niet aanwezig is in `sys.argv`.

```
qgis --code qgis_init.py --py-args myfile.tif -- otherfile.tif
```

U kunt `QCoreApplication.arguments()` gebruiken als u toegang wilt hebben tot elke parameter voor de opdrachtregel vanuit Python.

```
QgsApplication.instance().arguments()
```

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 standaardpaden 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 `initQgis()` aan te roepen.

```
qgs.initQgis()
```

Met QGIS geïnitialiseerd zijn we klaar om de rest van het script te schrijven. Tenslotte sluiten we af door de methode `exitQgis()` aan te roepen om de gegevensproviders en registratie van lagen uit het geheugen te verwijderen.

```
qgs.exitQgis()
```

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.

HOOFDSTUK 2

Projecten laden

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

```
from qgis.core import (
    Qgis,
    QgsProject,
    QgsPathResolver
)

from qgis.gui import (
    QgsLayerTreeMapCanvasBridge,
)
```

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())

# Load another project
project.read('testdata/01_project.qgs')
print(project.fileName())
```

```
testdata/01_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('testdata/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('testdata/my_new_qgis_project.qgs')
```

2.1 Slechte paden oplossen

Het kan gebeuren dat lagen die zijn geladen in het project worden verplaatst naar een andere locatie. Wanneer het project opnieuw wordt geladen zijn alle paden naar lagen verbroken. De klasse `QgsPathResolver` helpt u het pad naar de lagen in het project opnieuw te schrijven.

Zijn methode `setPathPreprocessor()` stelt u in staat een aangepaste functie pre-processor in te stellen voor het bewerken van paden en gegevensbronnen, voordat worden verbonden aan bestandsverwijzingen of bronnen van lagen.

De processor-functie moet één enkel argument tekenreeks accepteren (die het originele bestandspad of databron weergeeft) en geeft een verwerkte versie van dit pad terug. De functie pre-processor voor het pad wordt aangeroepen **voor** enige afhandeling van een slechts laag. Als meerdere pre-processors zijn ingesteld, zullen zij in een reeks worden aangeroepen, gebaseerd op de volgorde waarin zij origineel werden ingesteld.

Enkele gebruiksgevallen:

1. een verouderd pad vervangen:

```
def my_processor(path):
    return path.replace('c:/Users/ClintBarton/Documents/Projects', 'x:/
↪Projects/')

QgsPathResolver.setPathPreprocessor(my_processor)
```

2. het hostadres van een database vervangen door een nieuw:

```
def my_processor(path):
    return path.replace('host=10.1.1.115', 'host=10.1.1.116')

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. opgeslagen inloggegevens voor een database vervangen door nieuwe:

```
1 def my_processor(path):
2     path= path.replace("user='gis_team'", "user='team_awesome'")
3     path = path.replace("password='cats'", "password='g7as!m*'")
4     return path
5
6 QgsPathResolver.setPathPreprocessor(my_processor)
```

Op dezelfde wijze is een methode `setPathWriter()` beschikbaar als een functie om het pad te schrijven.

Een voorbeeld om het pad te vervangen door een variabele:

```
def my_processor(path):
  return path.replace('c:/Users/ClintBarton/Documents/Projects', '$projectdir$')

QgsPathResolver.setPathWriter(my_processor)
```

Beide methoden geven een `id` terug die kan worden gebruikt om de pre-processor of schrijver die zij hebben toegevoegd, te verwijderen Bekijk `removePathPreprocessor()` en `removePathWriter()`.

2.2 Vlaggen gebruiken om dingen te versnellen

In sommige gevallen, waarin u niet een volledig functioneel project zou hoeven te gebruiken, maar alleen toegang wilt voor een specifieke reden, kunnen vlaggen nuttig zijn. Een volledige lijst met vlaggen is beschikbaar onder `ProjectReadFlag`. Meerdere vlaggen mogen tegelijkertijd worden toegevoegd.

Als een voorbeeld: als we niet geven om de feitelijke lagen en gegevens en eenvoudigweg toegang tot een project willen (bijv. voor lay-out of instellingen voor 3D-weergave), kunnen we de vlag `DontResolveLayers` gebruiken om de stap voor het valideren van gegevens te omzeilen en te voorkomen dat het dialoogvenster voor slechte lagen verschijnt. Het volgende kan worden gedaan:

```
readflags = Qgis.ProjectReadFlags()
readflags |= Qgis.ProjectReadFlag.DontResolveLayers
project = QgsProject.instance()
project.read('C:/Users/ClintBarton/Documents/Projects/mysweetproject.qgs',␣
↪readflags)
```

Voor het toevoegen van meer vlaggen moet de Python Bitwise OR-operator (`|`) worden gebruikt.

HOOFDSTUK 3

Lagen laden

Hint: 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
)
```

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

To create and add a vector layer instance to the project, specify the layer's data source identifier. The data source identifier is a string and it is specific to each vector data provider. An optional layer name is used for identifying the layer in the *Layers* panel. It is important to check whether the layer has been loaded successfully. If it was not, an invalid layer instance is returned.

Voor een laag van GeoPackage:

```
# get the path to a geopackage
path_to_gpkg = "testdata/data/data.gpkg"
# append the layername part
gpkg_airports_layer = path_to_gpkg + "|layername=airports"
vlayer = QgsVectorLayer(gpkg_airports_layer, "Airports layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")
else:
    QgsProject.instance().addMapLayer(vlayer)
```

The quickest way to open and display a vector layer in QGIS is the `addVectorLayer()` method of the `QgisInterface` class:

```
vlayer = iface.addVectorLayer(gpkg_airports_layer, "Airports 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:

- The ogr provider from the GDAL library supports a wide variety of formats, also called drivers in GDAL speak. Examples are ESRI Shapefile, Geopackage, Flatgeobuf, Geojson, … For single-file formats the filepath usually suffices as uri. For geopackages or dxf, a pipe separated suffix allows to specify the layer to load.
  - for ESRI Shapefile:

    ```
    uri = "testdata/airports.shp"
    vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
    QgsProject.instance().addMapLayer(vlayer)
    ```

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

    ```
    uri = "testdata/data/data.gpkg|layername=airports"
    vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
    QgsProject.instance().addMapLayer(vlayer)
    ```

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

    ```
    uri = "testdata/sample.dxf|layername=entities|geometrytype=Polygon"
    vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
    QgsProject.instance().addMapLayer(vlayer)
    ```

- 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", "primary_key_
↪field")

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 = "file://{}/testdata/delimited_xy.csv?delimiter={}&xField={}&yField={}".
↪format(os.getcwd(), ";", "x", "y")
vlayer = QgsVectorLayer(uri, "layer name you like", "delimitedtext")
QgsProject.instance().addMapLayer(vlayer)
```

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 = "testdata/layers.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer name you like", "gpx")
QgsProject.instance().addMapLayer(vlayer)
```

- 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')
QgsProject.instance().addMapLayer(vlayer)
```

- MySQL op WKB gebaseerde geometrieën, via GDAL — 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" )
QgsProject.instance().addMapLayer(vlayer)
```

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

```
uri = "https://demo.mapserver.org/cgi-bin/wfs?service=WFS&version=2.0.0&
↪request=GetFeature&typename=ms:cities"
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")
```

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

```
import urllib

params = {
    'service': 'WFS',
    'version': '2.0.0',
    'request': 'GetFeature',
    'typename': 'ms:cities',
    'srsname': "EPSG:4326"
}
uri2 = 'https://demo.mapserver.org/cgi-bin/wfs?' + urllib.parse.unquote(urllib.
↪parse.urlencode(params))
```

Notitie: U kunt de databron van een bestaande laag wijzigen door `setDataSource()` aan te roepen op een instantie van `QgsVectorLayer`, zoals in het volgende voorbeeld:

```
uri = "https://demo.mapserver.org/cgi-bin/wfs?service=WFS&version=2.0.0&
↪request=GetFeature&typename=ms:cities"
provider_options = QgsDataProvider.ProviderOptions()
# Use project's transform context
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
provider_options.transformContext = QgsProject.instance().transformContext()
vlayer.setDataSource(uri, "layer name you like", "WFS", provider_options)

del(vlayer)
```

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 = "qgis-projects/python_cookbook/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(os.getcwd(), "testdata", "sublayers.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_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).

Een raster van PostGIS laden:

Rasters van PostGIS kunnen, soortgelijk aan vectors van PostGIS, aan een project worden toegevoegd met een tekenreeks URI. Het is efficiënt om een opnieuw te gebruiken woordenboek van tekenreeksen te maken voor de parameters van de verbinding voor de database. Dit maakt het eenvoudiger het woorden boek te bewerken voor de van toepassing zijnde verbinding. Het woordenboek wordt dan gecodeerd in een lege URI, met het provider metadata object 'postgresraster'. Daarna kan het raster worden toegevoegd aan het project.

```
uri_config = {
    # database parameters
    'dbname':'gis_db',       # The PostgreSQL database to connect to.
    'host':'localhost',      # The host IP address or localhost.
    'port':'5432',           # The port to connect on.
    'sslmode':QgsDataSourceUri.SslDisable, # SslAllow, SslPrefer, SslRequire,␣
↪SslVerifyCa, SslVerifyFull
    # user and password are not needed if stored in the authcfg or service
    'authcfg':'QconfigId',  # The QGIS athentication database ID holding␣
↪connection details.
    'service': None,          # The PostgreSQL service to be used for connection to␣
↪the database.
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    'username':None,         # The PostgreSQL user name.
    'password':None,         # The PostgreSQL password for the user.
    # table and raster column details
    'schema':'public',       # The database schema that the table is located in.
    'table':'my_rasters',    # The database table to be loaded.
    'geometrycolumn':'rast',# raster column in PostGIS table
    'sql':None,              # An SQL WHERE clause. It should be placed at the end␣
↪of the string.
    'key':None,              # A key column from the table.
    'srid':None,             # A string designating the SRID of the coordinate␣
↪reference system.
    'estimatedmetadata':'False', # A boolean value telling if the metadata is␣
↪estimated.
    'type':None,             # A WKT string designating the WKB Type.
    'selectatid':None,       # Set to True to disable selection by feature ID.
    'options':None,          # other PostgreSQL connection options not in this list.
    'enableTime': None,
    'temporalDefaultTime': None,
    'temporalFieldIndex': None,
    'mode':'2',              # GDAL 'mode' parameter, 2 unions raster tiles, 1 adds␣
↪tiles separately (may require user input)
}
# remove any NULL parameters
uri_config = {key:val for key, val in uri_config.items() if val is not None}
# get the metadata for the raster provider and configure the URI
md = QgsProviderRegistry.instance().providerMetadata('postgresraster')
uri = QgsDataSourceUri(md.encodeUri(uri_config))

# the raster can then be loaded into the project
rlayer = iface.addRasterLayer(uri.uri(False), "raster layer name", "postgresraster
↪")
```

Rasterlagen kunnen ook worden gemaakt vanuit een service voor WCS:

```
layer_name = 'modis'
url = "https://demo.mapserver.org/cgi-bin/wcs?identifier={}".format(layer_name)
rlayer = QgsRasterLayer(uri, '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 = "crs=EPSG:4326&format=image/png&layers=continents&styles&
↪url=https://demo.mapserver.org/cgi-bin/wms"
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()
```

HOOFDSTUK 4

Toegang tot de inhoudsopgave (TOC)

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

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

U kunt verschillende klassen gebruiken om toegang te verkrijgen tot alle geladen lagen in de inhoudsopgave en ze gebruiken om informatie op te halen:

- `QgsProject`
- `QgsLayerTreeGroup`

4.1 De klasse QgsProject

U kunt `QgsProject` gebruiken om informatie op te halen over de inhoudsopgave en alle geladen lagen.

U moet een `instance` van `QgsProject` maken en de methoden daarvan gebruiken om de geladen lagen op te halen.

De belangrijkste methode is `mapLayers()`. Het zal een woordenboek teruggeven van de geladen lagen:

```
layers = QgsProject.instance().mapLayers()
print(layers)
```

```
{'countries_89ae1b0f_f41b_4f42_bca4_caf55ddbe4b6': <QgsVectorLayer: 'countries'␣
↪(ogr)>}
```

De `keys` in het woordenboek zijn de unieke laag-ID's, terwijl de `values` de gerelateerde objecten zijn.

Het is nu rechtdoorzee om alle andere informatie over de lagen op te halen:

```
# list of layer names using list comprehension
l = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
# dictionary with key = layer name and value = layer object
layers_list = {}
for l in QgsProject.instance().mapLayers().values():
  layers_list[l.name()] = l

print(layers_list)
```

```
{'countries': <QgsVectorLayer: 'countries' (ogr)>}
```

U kunt ook de inhoudsopgave bevragen met de naam van de laag:

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

Notitie: Een lijst met alle overeenkomende lagen wordt teruggegeven, dus maken we een index met `[0]` om de eerste laag met zijn naam op te halen.

4.2 klasse QgsLayerTreeGroup

De boom met lagen is een klassieke boomstructuur, gebouwd uit knopen. Er zijn momenteel twee groepen knopen: groepsknopen (`QgsLayerTreeGroup`) en laagknopen (`QgsLayerTreeLayer`).

Notitie: Voor meer informatie kunt u deze blogposts van Martin Dobias lezen: Part 1 Part 2 Part 3

Toegang tot de boom met lagen van het project kan gemakkelijk worden verkregen met de methode `layerTreeRoot()` van de klasse `QgsProject`:

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

`root` is een groepsknoop en heeft *children*:

```
root.children()
```

Een lijst met directe kinderen wordt teruggegeven. Toegang tot kinderen van een subgroep zou moeten worden verkregen via hun eigen directe ouder.

We kunnen een van de kinderen ophalen:

```
child0 = root.children()[0]
print(child0)
```

```
<QgsLayerTreeLayer: countries>
```

Lagen kunnen ook worden opgehaald met hun (unieke) `id`:

```
ids = root.findLayerIds()
# access the first layer of the ids list
root.findLayer(ids[0])
```

En naar groepen kan ook worden gezocht met hun namen:

```
root.findGroup('Group Name')
```

`QgsLayerTreeGroup` heeft nog veel meer nuttige methoden die kunnen worden gebruikt om meer informatie op te halen over de inhoudsopgave:

```
# list of all the checked layers in the TOC
checked_layers = root.checkedLayers()
print(checked_layers)
```

```
[<QgsVectorLayer: 'countries' (ogr)>]
```

Laten we nu enkele lagen toevoegen aan de boom met lagen van het project. Er zijn twee manieren om dat te doen:

1. **Expliciete toevoeging** met de functies `addLayer()` of `insertLayer()`:

```
# create a temporary layer
layer1 = QgsVectorLayer("path_to_layer", "Layer 1", "memory")
# add the layer to the legend, last position
root.addLayer(layer1)
# add the layer at given position
root.insertLayer(5, layer1)
```

2. **Impliciete toevoeging**: omdat de boom met lagen van het project is verbonden met het register van de lagen is het voldoende om een laag toe te voegen aan het register met kaartlagen:

```
QgsProject.instance().addMapLayer(layer1)
```

U kunt gemakkelijk schakelen tussen `QgsVectorLayer` en `QgsLayerTreeLayer`:

```
node_layer = root.findLayer(country_layer.id())
print("Layer node:", node_layer)
print("Map layer:", node_layer.layer())
```

```
Layer node: <QgsLayerTreeLayer: countries>
Map layer: <QgsVectorLayer: 'countries' (ogr)>
```

Groepen kunnen worden toegevoegd met de methode `addGroup()`. In het voorbeeld hieronder, zal de eerste een groep toevoegen aan het einde van de inhoudsopgave, terwijl u met de laatste een andere groep kan toevoegen binnen een bestaande:

```
node_group1 = root.addGroup('Simple Group')
# add a sub-group to Simple Group
node_subgroup1 = node_group1.addGroup("I'm a sub group")
```

Er zijn vele nuttige methoden om knopen en groepen te verplaatsen.

Verplaatsen van een bestaande knoop wordt gedaan in drie stappen:

1. klonen van de bestaande knoop
2. verplaatsen van de gekloonde knoop naar de gewenste positie
3. verwijderen van de originele knoop

```
# clone the group
cloned_group1 = node_group1.clone()
# move the node (along with sub-groups and layers) to the top
root.insertChildNode(0, cloned_group1)
# remove the original node
root.removeChildNode(node_group1)
```

Het is iets meer *gecompliceerder* om een laag in de legenda te verplaatsen:

```
# get a QgsVectorLayer
vl = QgsProject.instance().mapLayersByName("countries")[0]
# create a QgsLayerTreeLayer object from vl by its id
myvl = root.findLayer(vl.id())
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
# clone the myvl QgsLayerTreeLayer object
myvlclone = myvl.clone()
# get the parent. If None (layer is not in group) returns ''
parent = myvl.parent()
# move the cloned layer to the top (0)
parent.insertChildNode(0, myvlclone)
# remove the original myvl
root.removeChildNode(myvl)
```

of om hem naar een bestaande groep te verplaatsen:

```
# get a QgsVectorLayer
vl = QgsProject.instance().mapLayersByName("countries")[0]
# create a QgsLayerTreeLayer object from vl by its id
myvl = root.findLayer(vl.id())
# clone the myvl QgsLayerTreeLayer object
myvlclone = myvl.clone()
# create a new group
group1 = root.addGroup("Group1")
# get the parent. If None (layer is not in group) returns ''
parent = myvl.parent()
# move the cloned layer to the top (0)
group1.insertChildNode(0, myvlclone)
# remove the QgsLayerTreeLayer from its parent
parent.removeChildNode(myvl)
```

Enkele andere methoden die kunnen worden gebruikt om groepen en lagen aan te passen:

```
node_group1 = root.findGroup("Group1")
# change the name of the group
node_group1.setName("Group X")
node_layer2 = root.findLayer(country_layer.id())
# change the name of the layer
node_layer2.setName("Layer X")
# change the visibility of a layer
node_group1.setItemVisibilityChecked(True)
node_layer2.setItemVisibilityChecked(False)
# expand/collapse the group view
node_group1.setExpanded(True)
node_group1.setExpanded(False)
```

HOOFDSTUK 5

Rasterlagen gebruiken

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

```
from qgis.core import (
    QgsRasterLayer,
    QgsProject,
    QgsPointXY,
    QgsRaster,
    QgsRasterShader,
    QgsColorRampShader,
    QgsSingleBandPseudoColorRenderer,
    QgsSingleBandColorDataRenderer,
    QgsSingleBandGrayRenderer,
)

from qgis.PyQt.QtGui import (
    QColor,
)
```

5.1 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
print(rlayer.width(), rlayer.height())
```

```
919 619
```

```
# get the extent of the layer as QgsRectangle
print(rlayer.extent())
```

```
<QgsRectangle: 20.06856808199999875 -34.27001076999999896, 20.83945284300000012 -
↪33.75077500700000144>
```

```
# get the extent of the layer as Strings
print(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
print(rlayer.rasterType())
```

```
0
```

```
 # get the total band count of the raster
print(rlayer.bandCount())
```

```
1
```

```
# get the first band name of the raster
print(rlayer.bandName(1))
```

```
Band 1: Height
```

```
# get all the available metadata as a QgsLayerMetadata object
print(rlayer.metadata())
```

```
<qgis._core.QgsLayerMetadata object at 0x13711d558>
```

5.2 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:

```
print(rlayer.renderer())
```

```
<qgis._core.QgsSingleBandGrayRenderer object at 0x7f471c1da8a0>
```

```
print(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`):

- `QgsHillshadeRenderer`
- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`

- QgsRasterContourRenderer
- 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.

5.2.1 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()
```

5.2.2 Multiband rasters

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

5.3 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())
```

```
{1: 323.0}
```

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}`

5.4 Bewerken van rastergegevens

U kunt een rasterlaag maken met de klasse `QgsRasterBlock`. Bijvoorbeeld om een rasterblok van 2x2 met één byte per pixel te maken:

```
block = QgsRasterBlock(Qgis.Byte, 2, 2)
block.setData(b'\xaa\xbb\xcc\xdd')
```

Rasterpixels kunnen worden overschreven dankzij de methode `writeBlock()`. Om bestaande rastergegevens te overschrijven op positie 0,0 van het 2x2-blok:

```
provider = rlayer.dataProvider()
provider.setEditable(True)
provider.writeBlock(block, 1, 0, 0)
provider.setEditable(False)
```

HOOFDSTUK 6

Vectorlagen gebruiken

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

```
from qgis.core import (
  QgsApplication,
  QgsDataSourceUri,
  QgsCategorizedSymbolRenderer,
  QgsClassificationRange,
  QgsPointXY,
  QgsProject,
  QgsExpression,
  QgsField,
  QgsFields,
  QgsFeature,
  QgsFeatureRequest,
  QgsFeatureRenderer,
  QgsGeometry,
  QgsGraduatedSymbolRenderer,
  QgsMarkerSymbol,
  QgsMessageLog,
  QgsRectangle,
  QgsRendererCategory,
  QgsRendererRange,
  QgsSymbol,
  QgsVectorDataProvider,
  QgsVectorLayer,
  QgsVectorFileWriter,
  QgsWkbTypes,
  QgsSpatialIndex,
  QgsVectorLayerUtils
)

from qgis.core.additions.edit import edit

from qgis.PyQt.QtGui import (
    QColor,
)
```

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

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

6.1 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`:

```
vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports␣
↪layer", "ogr")
for field in vlayer.fields():
    print(field.name(), field.typeName())
```

```
fid Integer64
id Integer64
scalerank Integer64
featurecla String
type String
name String
abbrev String
location String
gps_code String
iata_code String
wikipedia String
natlscale Real
```

De methoden `displayField()` en `mapTipTemplate()` verschaffen informatie over het veld en de gebruikte sjabloon op de tab maptips.

Wanneer u een vectorlaag laadt, wordt altijd een veld gekozen door QGIS als de `Weergavenaam`, terwijl de `HTML kaarttip` standaard leeg is. Met deze methoden kunt u gemakkelijk beide verkrijgen:

```
vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports␣
↪layer", "ogr")
print(vlayer.displayField())
```

```
name
```

Notitie: Als u de `Weergavenaam` wijzigt van een veld naar een expressie, moet u `displayExpression()` gebruiken in plaats van `displayField()`.

6.2 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" is a QgsVectorLayer instance
layer = iface.activeLayer()
features = layer.getFeatures()

for feature in features:
    # retrieve every feature with its geometry and attributes
    print("Feature ID: ", feature.id())
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    # 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)
    # for this test only print the first feature
    break
```

```
Feature ID:  1
Point:  <QgsPointXY: POINT(7 45)>
[1, 'First feature']
```

6.3 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.

Soms kan het nuttig zijn om programmatisch objecten te selecteren of om de standaardkleur te wijzigen.

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()
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
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
feature = next(layer.getFeatures())
if feature:
    selected_fid.append(feature.id())

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

De selectie opheffen:

```
layer.removeSelection()
```

6.3.1 Toegang tot attributen

Naar attributen kan worden verwezen door middel van hun naam:

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

```
First feature
```

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 tweede attribuut te krijgen:

```
print(feature[1])
```

```
First feature
```

6.3.2 Itereren over geselecteerde objecten

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

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

6.3.3 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
    pass
```

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):
    print(feature)
```

```
<qgis._core.QgsFeature object at 0x7f9b78590948>
<qgis._core.QgsFeature object at 0x7faef5881670>
```

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])
```

6.4 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')
```

```
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()
```

6.4.1 Objecten toevoegen

Maak enkele instances `QgsFeature` en geef daar een lijst van door aan de methode `QgsVectorDataProvider addFeatures()` van de provider. Het zal twee waarden teruggeven: resultaat (`True` of `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')
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

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

6.4.2 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])
```

6.4.3 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.)

6.4.4 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 QMetaType

feat1 = feat2 = QgsFeature(layer.fields())
fid = 99
feat1.setId(fid)

# 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
geometry = QgsGeometry.fromWkt("POINT(7 45)")
layer.changeGeometry(fid, geometry)
# update an attribute with given field index (int) to a given value
fieldIndex =1
value ='My new name'
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QMetaType.Type.QString))
# 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()
  # ... tell the user that there was a problem
  # and 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.

6.4.5 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 QMetaType

if caps & QgsVectorDataProvider.AddAttributes:
    res = layer.dataProvider().addAttributes(
        [QgsField("mytext", QMetaType.Type.QString),
        QgsField("myint", QMetaType.Type.Int)])

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

```
# Alternate methods for removing fields
# first create temporary fields to be removed (f1-3)
layer.dataProvider().addAttributes([QgsField("f1", QMetaType.Type.Int),
                                    QgsField("f2", QMetaType.Type.Int),
                                    QgsField("f3", QMetaType.Type.Int)])
layer.updateFields()
count=layer.fields().count() # count of layer fields
ind_list=list((count-3, count-2)) # create list

# remove a single field with an index
layer.dataProvider().deleteAttributes([count-1])

# remove multiple fields with a list of indices
layer.dataProvider().deleteAttributes(ind_list)
```

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*.

6.5 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:

- ruimtelijke index maken 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.addFeature(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))
```

U kunt ook de ruimtelijke index `QgsSpatialIndexKDBush` gebruiken. Deze index is soortgelijk aan de *standaard* `QgsSpatialIndex` maar:

- ondersteunt **alleen** zelfstandige puntobjecten
- is **statisch** (geen aanvullende objecten kunnen aan de index worden toegevoegd na het construeren)
- is **veel sneller!**
- maakt direct ophalen van de originele punten van het object mogelijk, zonder aanvullende verzoeken aan het object nodig te hebben
- ondersteunt ware *op afstand gebaseerde* zoekacties, d.i. geeft alle punten terug binnen een straal vanaf een zoekpunt

6.6 De klasse QgsVectorLayerUtils

De klasse `QgsVectorLayerUtils` bevat enkele zeer nuttige methoden die u kunt gebruiken met vectorlagen.

Bijvoorbeeld de methode `createFeature()` prepareert een `QgsFeature` om te worden toegevoegd aan een vectorlaag, met behoud van alle eventueel bestaande beperkingen en standaardwaarden van elk veld:

```
vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports␣
↪layer", "ogr")
feat = QgsVectorLayerUtils.createFeature(vlayer)
```

De methode `getValues()` stelt u in staat snel de waarden van een veld of expressie op te halen:

```
vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports␣
↪layer", "ogr")
# select only the first feature to make the output shorter
vlayer.selectByIds([1])
val = QgsVectorLayerUtils.getValues(vlayer, "NAME", selectedOnly=True)
print(val)
```

```
(['Sahnewal'], True)
```

6.7 Vectorlagen maken

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

- de klasse `QgsVectorFileWriter`: Een nuttige klasse voor het schrijven van vectorbestanden naar schijf, met ofwel een statische aanroep naar `writeAsVectorFormatV3()` die de gehele vectorlaag opslaat of het maken van een instantie van de klasse en aanroepen uitvoeren naar de geërfde `addFeature()`. Deze klasse ondersteunt alle vectorindelingen die GDAL 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 GDAL vector gegevenssets (`ogr`), databases (`postgres`, `spatialite`, `mysql`, `mssql`) en meer (`wfs`, `gpx`, `delimitedtext`...).

6.7.1 Vanuit een instance van `QgsVectorFileWriter`

```
# SaveVectorOptions contains many settings for the writer process
save_options = QgsVectorFileWriter.SaveVectorOptions()
transform_context = QgsProject.instance().transformContext()
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
                                                  "testdata/my_new_file.gpkg",
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
else:
  print(error)
```

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "ESRI Shapefile"
save_options.fileEncoding = "UTF-8"
transform_context = QgsProject.instance().transformContext()
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
                                                  "testdata/my_new_shapefile",
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
else:
  print(error)
```

```
# Write to an ESRI GDB file
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "FileGDB"
# if no geometry
save_options.overrideGeometryType = QgsWkbTypes.Unknown
save_options.actionOnExistingFile = QgsVectorFileWriter.CreateOrOverwriteLayer
save_options.layerName = 'my_new_layer_name'
transform_context = QgsProject.instance().transformContext()
gdb_path = "testdata/my_example.gdb"
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
                                                  gdb_path,
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
  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, QMetaType.Type.QString)
    else:
      return self.layer.fields()[idx]

converter = ESRIValueConverter(vlayer, LIST_FIELD_NAME)
opts = QgsVectorFileWriter.SaveVectorOptions()
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.

6.7.2 Direct uit objecten

```
from qgis.PyQt.QtCore import QMetaType

# define fields for feature attributes. A QgsFields object is needed
fields = QgsFields()
fields.append(QgsField("first", QMetaType.Type.Int))
fields.append(QgsField("second", QMetaType.Type.QString))

""" 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. field map
3. geometry type - from WKBTYPE enum
4. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem)
5. coordinate transform context
6. save options (driver name for the output file, encoding etc.)
"""

crs = QgsProject.instance().crs()
transform_context = QgsProject.instance().transformContext()
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "ESRI Shapefile"
save_options.fileEncoding = "UTF-8"

writer = QgsVectorFileWriter.create(
  "testdata/my_new_shapefile.shp",
  fields,
  QgsWkbTypes.Point,
  crs,
  transform_context,
  save_options
)

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

6.7.3 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 QMetaType

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

# add fields
pr.addAttributes([QgsField("name", QMetaType.Type.QString),
                  QgsField("age",  QMetaType.Type.Int),
                  QgsField("size", QMetaType.Type.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(["Johnny", 2, 0.3])
pr.addFeatures([fet])

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

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())
```

```
fields: 3
features: 1
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
extent: 10.0 10.0 10.0 10.0
F: 1 ['Johnny', 2, 0.3] <QgsPointXY: POINT(10 10)>
```

6.8 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())
```

```
Type: singleSymbol
```

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

Type	Klasse	Beschrijving
singleSymbol	QgsSingleSymbolRender	Rendert alle objecten met hetzelfde symbool
categorizedSyn	QgsCategorizedSymbolR	Rendert objecten door een ander symbool voor elke categorie te gebruiken
graduatedSymb	QgsGraduatedSymbolRen	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())
```

```
['nullSymbol', 'singleSymbol', 'categorizedSymbol', 'graduatedSymbol',
↪'RuleRenderer', 'pointDisplacement', 'pointCluster', 'mergedFeatureRenderer',
↪'invertedPolygonRenderer', 'heatmapRenderer', '25dRenderer', 'embeddedSymbol']
```

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

```
renderer.dump()
```

```
SINGLE: MARKER SYMBOL (1 layers) color 190,207,80,255
```

6.8.1 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())
```

```
{'angle': '0', 'cap_style': 'square', 'color': '255,0,0,255,rgb:1,0,0,1',
↪'horizontal_anchor_point': '1', 'joinstyle': 'bevel', 'name': 'square', 'offset
↪': '0,0', 'offset_map_unit_scale': '3x:0,0,0,0,0,0', 'offset_unit': 'MM',
↪'outline_color': '35,35,35,255,rgb:0.13725490196078433,0.13725490196078433,0.
↪13725490196078433,1', 'outline_style': 'solid', 'outline_width': '0', 'outline_
↪width_map_unit_scale': '3x:0,0,0,0,0,0', 'outline_width_unit': 'MM', 'scale_
↪method': 'diameter', 'size': '2', 'size_map_unit_scale': '3x:0,0,0,0,0,0', '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()
```

6.8.2 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

```
categorized_renderer = QgsCategorizedSymbolRenderer()
# Add a few categories
cat1 = QgsRendererCategory('1', QgsMarkerSymbol(), 'category 1')
cat2 = QgsRendererCategory('2', QgsMarkerSymbol(), 'category 2')
categorized_renderer.addCategory(cat1)
categorized_renderer.addCategory(cat2)

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

```
1: category 1 :: <qgis._core.QgsMarkerSymbol object at 0x7f378ffcd9d8>
2: category 2 :: <qgis._core.QgsMarkerSymbol object at 0x7f378ffcd9d8>
```

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()`.

6.8.3 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

```
graduated_renderer = QgsGraduatedSymbolRenderer()
# Add a few categories
graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class 0-
↪100', 0, 100), QgsMarkerSymbol()))
graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class␣
↪101-200', 101, 200), QgsMarkerSymbol()))

for ran in graduated_renderer.ranges():
    print("{} - {}: {} {}".format(
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        ran.symbol()
      ))
```

```
0.0 - 100.0: class 0-100 <qgis._core.QgsMarkerSymbol object at 0x7f8bad281b88>
101.0 - 200.0: class 101-200 <qgis._core.QgsMarkerSymbol object at 0x7f8bad281b88>
```

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("testdata/data/data.gpkg|layername=airports",
↪"Airports layer", "ogr")
myTargetField = 'scalerank'
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)
myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
↪"EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
```

6.8.4 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:

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

```
0: SimpleMarker
```

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)
```

```
AnimatedMarker
EllipseMarker
FilledMarker
FontMarker
GeometryGenerator
MaskMarker
RasterMarker
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 size() en angle() alleen beschikbaar voor symboollagen voor markeringen en width() 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)
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        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):
    super().__init__("FooMarker", "My new Foo marker", QgsSymbol.Marker)

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

fslmetadata = FooSymbolLayerMetadata()
```

```
QgsApplication.symbolLayerRegistry().addSymbolLayerType(fslmetadata)
```

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.

6.8.5 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):
    super().__init__("RandomRenderer")
    self.syms = syms if syms else [
      QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point)),
      QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point))
    ]

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

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

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

  def usedAttributes(self, context):
    return []

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

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

```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```

class RandomRendererWidget(QgsRendererWidget):
  def __init__(self, layer, style, renderer):
    super().__init__(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.colorChanged.connect(self.setColor1)

  def setColor1(self):
    color = self.btn1.color()
    if not color.isValid(): return
    self.r.syms[0].setColor(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 ander 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):
    super().__init__("RandomRenderer", "Random renderer")

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

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

rrmetadata = RandomRendererMetadata()
```

```
QgsApplication.rendererRegistry().addRenderer(rrmetadata)
```

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).

6.9 Meer onderwerpen

TODO:

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

HOOFDSTUK 7

Afhandeling van geometrie

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

```
from qgis.core import (
  QgsGeometry,
  QgsGeometryCollection,
  QgsPoint,
  QgsPointXY,
  QgsWkbTypes,
  QgsProject,
  QgsFeatureRequest,
  QgsVectorLayer,
  QgsDistanceArea,
  QgsUnitTypes,
  QgsCoordinateTransform,
  QgsCoordinateReferenceSystem
)
```

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.

7.1 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())
```

7.2 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.

```
print(gPnt.wkbType())
# output: 1
print(gLine.wkbType())
# output: 2
print(gPolygon.wkbType())
# output: 3
```

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

```
print(gLine.type())
# output: 1
```

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'
```

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()`.

Het is mogelijk door alle delen van een geometrie te gaan, ongeacht het type geometrie. Bijv.

```
geom = QgsGeometry.fromWkt( 'MultiPoint( 0 0, 1 1, 2 2)' )
for part in geom.parts():
  print(part.asWkt())
```

```
Point (0 0)
Point (1 1)
Point (2 2)
```

```
geom = QgsGeometry.fromWkt( 'LineString( 0 0, 10 10 )' )
for part in geom.parts():
  print(part.asWkt())
```

```
LineString (0 0, 10 10)
```

```
gc = QgsGeometryCollection()
gc.fromWkt('GeometryCollection( Point(1 2), Point(11 12), LineString(33 34, 44 45))
↪')
print(gc[1].asWkt())
```

```
Point (11 12)
```

Het is ook mogelijk elk deel van de geometrie aan te passen met de methode `QgsGeometry.parts()`.

```
geom = QgsGeometry.fromWkt( 'MultiPoint( 0 0, 1 1, 2 2)' )
for part in geom.parts():
  part.transform(QgsCoordinateTransform(
    QgsCoordinateReferenceSystem("EPSG:4326"),
    QgsCoordinateReferenceSystem("EPSG:3111"),
    QgsProject.instance())
  )
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
print(geom.asWkt())
```

```
MultiPoint ((-10334726.79314758814871311 -5360105.10101194866001606),(-10462133.
↪82917747274041176 -5217484.34365733992308378),(-10589398.51346861757338047 -
↪5072020.35880533326417208))
```

7.3 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())
```

```
Zambia
Area:  62.82279065343119
Perimeter:  50.65232014052552
Zimbabwe
Area:  33.41113559136517
Perimeter:  26.608288555013935
```

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))
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
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))
```

```
Zambia
Perimeter (m): 5539361.250294601
Area (m2): 751989035032.9031
Area (km2): 751989.0350329031
Zimbabwe
Perimeter (m): 2865021.3325076113
Area (m2): 389267821381.6008
Area (km2): 389267.8213816008
```

Als alternatief zou u misschien de afstand 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 algoritmes 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

HOOFDSTUK 8

Ondersteuning van projecties

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

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

8.1 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

```
# EPSG 4326 is allocated for WGS84
crs = QgsCoordinateReferenceSystem("EPSG:4326")
print(crs.isValid())
```

```
True
```

  QGIS ondersteunt verschillende identificaties voor CRS met de volgende indelingen:

  - `EPSG:<code>` — ID toegewezen door de organisatie EPSG - afgehandeld met `createFromOgcWms()`
  - `POSTGIS:<srid>`— ID gebruikt in databases van PostGIS - afgehandeld met `createFromSrid()`
  - `INTERNAL:<srsid>` — ID gebruikt in de interne database van QGIS - afgehandeld met `createFromSrsId()`
  - `PROJ:<proj>` - afgehandeld met `createFromProj()`
  - `WKT:<wkt>` - afgehandeld met `createFromWkt()`

Indien geen voorvoegsel is gespecificeerd, wordt definitie WKT aangenomen.

- 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",0.0174532925199433295],' \
      'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
print(crs.isValid())
```

```
True
```

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

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

```
True
```

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("EPSG: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("Proj String:", crs.toProj())
# 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: EPSG:7030
Proj String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

8.2 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("EPSG:4326")     # WGS 84
crsDest = QgsCoordinateReferenceSystem("EPSG:32633")  # WGS 84 / UTM zone 33N
transformContext = QgsProject.instance().transformContext()
xform = QgsCoordinateTransform(crsSrc, crsDest, transformContext)

# 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 4.99999999999999911)>
```

HOOFDSTUK 9

Het kaartvenster gebruiken

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

```
from qgis.PyQt.QtGui import (
    QColor,
)

from qgis.PyQt.QtCore import Qt, QRectF

from qgis.PyQt.QtWidgets import QMenu

from qgis.core import (
    QgsVectorLayer,
    QgsPoint,
    QgsPointXY,
    QgsProject,
    QgsGeometry,
    QgsMapRendererJob,
    QgsWkbTypes,
)

from qgis.gui import (
    QgsMapCanvas,
    QgsVertexMarker,
    QgsMapCanvasItem,
    QgsMapMouseEvent,
    QgsRubberBand,
)
```

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

9.1 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 en de lijst met lagen voor het kaartvenster.

```
1 vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports␣
  ↪layer", "ogr")
2 if not vlayer.isValid():
3     print("Layer failed to load!")
4
5 # add layer to the registry
6 QgsProject.instance().addMapLayer(vlayer)
7
8 # set extent to the extent of our layer
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
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.

9.2 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, QgsWkbTypes.LineGeometry)  # line
points = [QgsPoint(-100, 45), QgsPoint(10, 60), QgsPoint(120, 45)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Een polygoon weregeven

```
r = QgsRubberBand(canvas, QgsWkbTypes.PolygonGeometry)  # 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.

9.3 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)
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        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 laag in het nieuw gemaakte kaartvenster renderen

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

9.3.1 Een object selecteren met QgsMapToolIdentifyFeature

U kunt het kaartgereedschap `QgsMapToolIdentifyFeature` gebruiken om de gebruiker te vragen een object te selecteren dat moet worden verzonden naar een later aan te roepen functie.

```
def callback(feature):
  """Code called when the feature is selected by the user"""
  print("You clicked on feature {}".format(feature.id()))

canvas = iface.mapCanvas()
feature_identifier = QgsMapToolIdentifyFeature(canvas)

# indicates the layer on which the selection will be done
feature_identifier.setLayer(vlayer)

# use the callback as a slot triggered when the user identifies a feature
feature_identifier.featureIdentified.connect(callback)

# activation of the map tool
canvas.setMapTool(feature_identifier)
```

9.3.2 Items toevoegen aan het contextmenu van het kaartvenster

Interactie met het kaartvenster kan ook worden uitgevoerd via items die zou hebben toegevoegd tot het contextmenu daarvan met het signaal `contextMenuAboutToShow`.

De volgende code voegt de actie *My menu* ► *My Action* naast de standaard items in als u met rechts klikt in het kaartvenster.

```
# a slot to populate the context menu
def populateContextMenu(menu: QMenu, event: QgsMapMouseEvent):
    subMenu = menu.addMenu('My Menu')
    action = subMenu.addAction('My Action')
    action.triggered.connect(lambda *args:
                             print(f'Action triggered at {event.x()},{event.y()}'))

canvas.contextMenuAboutToShow.connect(populateContextMenu)
canvas.show()
```

9.4 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, QgsWkbTypes.PolygonGeometry)
    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(QgsWkbTypes.PolygonGeometry)

  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(QgsWkbTypes.PolygonGeometry)
    if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
      return

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

    self.rubberBand.addPoint(point1, False)
    self.rubberBand.addPoint(point2, False)
    self.rubberBand.addPoint(point3, False)
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    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()
```

9.5 Aangepaste items voor het kaartvenster schrijven

Hier is een voorbeeld van een aangepast item voor het kaartvenster dat een cirkel tekent:

```
class CircleCanvasItem(QgsMapCanvasItem):
  def __init__(self, canvas):
    super().__init__(canvas)
    self.center = QgsPoint(0, 0)
    self.size   = 100

  def setCenter(self, center):
    self.center = center

  def center(self):
    return self.center

  def setSize(self, size):
    self.size = size

  def size(self):
    return self.size

  def boundingRect(self):
    return QRectF(self.center.x() - self.size/2,
      self.center.y() - self.size/2,
      self.center.x() + self.size/2,
      self.center.y() + self.size/2)

  def paint(self, painter, option, widget):
    path = QPainterPath()
    path.moveTo(self.center.x(), self.center.y());
    path.arcTo(self.boundingRect(), 0.0, 360.0)
    painter.fillPath(path, QColor("red"))


# Using the custom item:
item = CircleCanvasItem(iface.mapCanvas())
item.setCenter(QgsPointXY(200,200))
item.setSize(80)
```

HOOFDSTUK 10

Kaart renderen en afdrukken

Hint: De codesnippers op deze pagina hebben de volgende import nodig:

```
import os

from qgis.core import (
    QgsGeometry,
    QgsMapSettings,
    QgsPrintLayout,
    QgsMapSettings,
    QgsMapRendererParallelJob,
    QgsLayoutItemLabel,
    QgsLayoutItemLegend,
    QgsLayoutItemMap,
    QgsLayoutItemPolygon,
    QgsLayoutItemScaleBar,
    QgsLayoutExporter,
    QgsLayoutItem,
    QgsLayoutPoint,
    QgsLayoutSize,
    QgsUnitTypes,
    QgsProject,
    QgsFillSymbol,
    QgsAbstractValidityCheck,
    check,
)

from qgis.PyQt.QtGui import (
    QPolygonF,
    QColor,
)

from qgis.PyQt.QtCore import (
    QPointF,
    QRectF,
    QSize,
)
```

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`.

10.1 Eenvoudig renderen

Het renderen wordt gedaan door een object `QgsMapSettings` te maken om de instellingen voor renderen te definiëren, en dan een `QgsMapRendererJob` te construeren met deze instellingen. 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")

vlayer = iface.activeLayer()
settings = QgsMapSettings()
settings.setLayers([vlayer])
settings.setBackgroundColor(QColor(255, 255, 255))
settings.setOutputSize(QSize(800, 600))
settings.setExtent(vlayer.extent())

render = QgsMapRendererParallelJob(settings)

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

render.finished.connect(finished)

# Start the rendering
render.start()

# The following loop is not normally required, we
# are using it here because this is a standalone example.
from qgis.PyQt.QtCore import QEventLoop
loop = QEventLoop()
render.finished.connect(loop.quit)
loop.exec_()
```

10.2 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.

```
layers = [iface.activeLayer()]
settings = QgsMapSettings()
settings.setLayers(layers)
settings.setDestinationCrs(layers[0].crs())
```

10.3 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, SVG, rasterafbeeldingen of direct worden afgedrukt op een printer.

De lay-out bestaat uit een aantal klassen. Zij maken allemaal deel uit van de bronbibliotheek. De toepassing QGIS heeft een handige gebruikersinterface voor de plaatsing van de elementen, hoewel die niet beschikbaar is in de bibliotheek van de gebruikersinterface. Als u nog niet bekend bent met Qt Graphics View framework, wordt u aangeraden om nu de documentatie te bekijken, omdat de lay-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:

```
project = QgsProject.instance()
layout = QgsPrintLayout(project)
layout.initializeDefaults()
```

Dit initialiseert de lay-out met enkele standaard instellingen, in het bijzonder een lege pagina A4 aan de lay-out. U kunt lay-outs maken zonder de methode `initializeDefaults()` aan te roepen, maar u dient zelf het toevoegen van pagina's aan de lay-out te regelen.

De vorige code maakt een "tijdelijke" lay-out die niet zichtbaar is in de GUI. Het kan handig zijn om bijvoorbeeld snel enkele items toe te voegen en te exporteren, zonder het project aan te passen of deze wijzigingen aan de gebruiker te laten zien. Als u de lay-out wilt opslaan/herstellen naast het project en beschikbaar zijn in Lay-out beheren, voeg dan toe:

```
layout.setName("MyLayout")
project.layoutManager().addLayout(layout)
```

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.

- map — Hier maken we een kaart van een aangepaste grootte en renderen het huidige kaartvenster

```
map = QgsLayoutItemMap(layout)
# Set map item position and size (by default, it is a 0 width/0 height item␣
↪placed at 0,0)
map.attemptMove(QgsLayoutPoint(5,5, QgsUnitTypes.LayoutMillimeters))
map.attemptResize(QgsLayoutSize(200,200, QgsUnitTypes.LayoutMillimeters))
# Provide an extent to render
map.zoomToExtent(iface.mapCanvas().extent())
layout.addLayoutItem(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.addLayoutItem(label)
```

- legenda

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

- schaalbalk

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

- 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.addLayoutItem(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)
```

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.

10.3.1 Geldigheid lay-out controleren

Een lay-out is gemaakt uit een set van onderling verbonden items en het kan gebeuren dat deze verbindingen defect raken tijdens aanpassingen (een legenda die is verbonden met een verwijderde kaart, een afbeeldingsitem met een ontbrekend bronbestand,…) of u wilt misschien aangepaste beperkingen toepassen op de items van de lay-out. De klasse `QgsAbstractValidityCheck` helpt u dit te bereiken.

Een basiscontrole ziet eruit als dit:

```
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def my_layout_check(context, feedback):
  results = ...
  return results
```

Hier is een controle die een waarschuwing opwerpt, iedere keer als een kaartitem van een lay-out is ingesteld op de projectie Web Mercator:

```
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def layout_map_crs_choice_check(context, feedback):
  layout = context.layout
  results = []
  for i in layout.items():
    if isinstance(i, QgsLayoutItemMap) and i.crs().authid() == 'EPSG:3857':
      res = QgsValidityCheckResult()
      res.type = QgsValidityCheckResult.Warning
      res.title = 'Map projection is misleading'
      res.detailedDescription = 'The projection for the map item {} is set to <i>
↪Web Mercator (EPSG:3857)</i> which misrepresents areas and shapes. Consider␣
↪using an appropriate local projection instead.'.format(i.displayName())
      results.append(res)

  return results
```

En hier is een meer complexer voorbeeld, dat een waarschuwing opwerpt als een kaartitem is ingesteld op een CRS dat alleen geldig is buiten het in dat kaartitem weergegeven bereik:

```
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def layout_map_crs_area_check(context, feedback):
    layout = context.layout
    results = []
    for i in layout.items():
        if isinstance(i, QgsLayoutItemMap):
            bounds = i.crs().bounds()
            ct = QgsCoordinateTransform(QgsCoordinateReferenceSystem('EPSG:4326'),␣
↪i.crs(), QgsProject.instance())
            bounds_crs = ct.transformBoundingBox(bounds)

            if not bounds_crs.contains(i.extent()):
                res = QgsValidityCheckResult()
                res.type = QgsValidityCheckResult.Warning
                res.title = 'Map projection is incorrect'
                res.detailedDescription = 'The projection for the map item {} is␣
↪set to \'{}\', which is not valid for the area displayed within the map.'.
↪format(i.displayName(), i.crs().authid())
                results.append(res)

    return results
```

10.3.2 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 `exportToSvg()` of `exportToImage()` in het geval dat u wilt exporteren naar een SVG of afbeeldingsbestand in plaats van een bestand PDF.

10.3.3 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.

HOOFDSTUK 11

Expressies, filteren en waarden berekenen

Hint: 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
)
```

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`

11.1 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 of file')
```

11.2 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.

11.2.1 Basisexpressies

Deze basis expressie evalueert een eenvoudige rekenkundige bewerking:

```
exp = QgsExpression('2 * 3')
print(exp)
print(exp.evaluate())
```

```
<QgsExpression: '2 * 3'>
6
```

Expressie mag ook gebruikt worden voor vergelijking, evalueert tot 1 (`True`) of 0 (`False`)

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

11.2.2 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)
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 QMetaType

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

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```

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'])
```

```
876.5
```

11.2.3 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)

```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

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

print(matches)
```

```
7
```

11.3 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())
```

HOOFDSTUK 12

Instellingen lezen en opslaan

Hint: 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
)
```

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:

- **globale instellingen** — zij zijn gebonden aan de gebruiker van een bepaalde machine. QGIS slaat zelf heel veel globale instellingen op, bijvoorbeeld, grootte van het hoofdvenster of standaard tolerantie voor snappen. Instellingen worden afgehandeld door de klasse `QgsSettings` class, door bijvoorbeeld de methoden `setValue()` en `value()`.

  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)
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
  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.

Voor een methode om de standaardwaarden voor de globale instellingen vooraf te configureren via het bestand `qgis_global_settings.ini`, bekijk deploying_organization voor meer details.

- **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.writeEntryDouble("myplugin", "mydouble", 0.01)
proj.writeEntryBool("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 veel gegevenstypen (integer, string, list), 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 opgegeven 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")
```

HOOFDSTUK 13

Communiceren met de gebruiker

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

```
from qgis.core import (
    QgsMessageLog,
    QgsGeometry,
)

from qgis.gui import (
    QgsMessageBar,
)

from qgis.PyQt.QtWidgets import (
    QSizePolicy,
    QPushButton,
    QDialog,
    QGridLayout,
    QDialogButtonBox,
)
```

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.

13.1 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)
```

```
Messages(2): Error : I'm sorry Dave, I'm afraid I can't do that
```

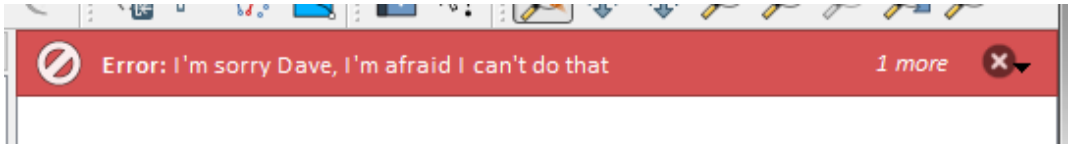


Fig. 13.1: QGIS berichtenbalk

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)
```

```
Messages(2): Ooops : The plugin is not working as it should
```

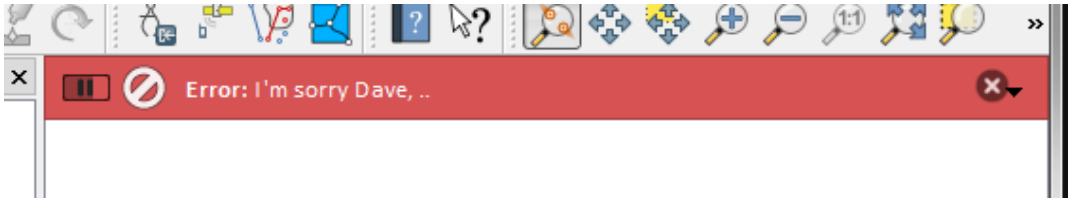


Fig. 13.2: QGIS berichtenbalk met timer

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

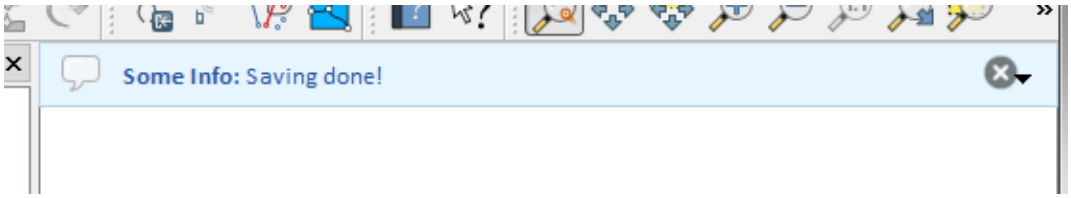


Fig. 13.3: QGIS berichtenbalk (info)

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)
```

```
Messages(1): Missing Layers : Show Me
```

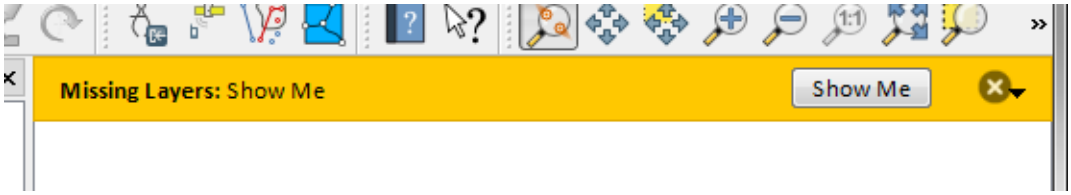


Fig. 13.4: QGIS berichtenbalk met een knop

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()
```

Fig. 13.5: QGIS berichtenbalk in aangepast dialoogvenster

13.2 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()
```

```
Messages(0): Doing something boring...
```

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

```
vlayer = iface.activeLayer()

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()
```

13.3 Loggen

Er zijn in QGIS drie verschillende soorten loggen beschikbaar om alle informatie over het uitvoeren van uw code te loggen en op te slaan. Elk heeft zijn eigen specifieke locatie voor de uitvoer. Bepaal eerst de juiste manier van loggen voor uw doel:

- `QgsMessageLog` is voor het communiceren met gebruikers van problemen. De uitvoer van het QgsMessageLog wordt weergegeven in het paneel Logboekberichten.
- De ingebouwde module van Python voor **logging** is voor het debuggen op het niveau van de QGIS Python API (PyQGIS). Het wordt aanbevolen aan scriptontwikkelaars in Python die hun code voor Python moeten debuggen, bijv. object-ID's of geometrieën
- `QgsLogger` is voor berichten voor *QGIS intern* debuggen / ontwikkelaars d.i. als u vermoedt dat zij worden veroorzaakt door een beschadigde code). Berichten zijn alleen zichtbaar in ontwikkelversies van QGIS.

Voorbeelden voor de verschillende typen van loggen worden weergegeven in de volgende gedeelten hieronder.

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

13.3.1 QgsMessageLog

```
# 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)
```

```
MyPlugin(0): Your plugin code has been executed correctly
(1): Your plugin code might have some problems
(2): Your plugin code has crashed!
```

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

13.3.2 De ingebouwde module voor loggen in Python

```
import logging
formatter = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
logfilename=r'c:\temp\example.log'
logging.basicConfig(filename=logfilename, level=logging.DEBUG, format=formatter)
logging.info("This logging info text goes into the file")
logging.debug("This logging debug text goes into the file as well")
```

De methode basicConfig configureert de basis instelling van het loggen. In de bovenstaande code worden de bestandsnaam, het niveau van loggen en de indeling gedefinieerd. De bestandsnaam verwijst naar waar het logbestand moet worden weggeschreven, het niveau van loggen definieert welke niveaus moeten worden uitgevoerd en de indeling definieert de indeling waarin elk bericht wordt uitgevoerd.

```
2020-10-08 13:14:42,998 - root - INFO - This logging text goes into the file
2020-10-08 13:14:42,998 - root - DEBUG - This logging debug text goes into the␣
↪file as well
```

Als u het logbestand wilt verwijderen, elke keer als u uw script uitvoert, kunt u iets doen als:

```
if os.path.isfile(logfilename):
    with open(logfilename, 'w') as file:
        pass
```

Meer bronnen over hoe de mogelijkheden voor loggen in Python te gebruiken zijn beschikbaar op:

- https://docs.python.org/3/library/logging.html
- https://docs.python.org/3/howto/logging.html
- https://docs.python.org/3/howto/logging-cookbook.html

Waarschuwing: Onthoud dat loggen zonder dat naar en bestand te sturen door een bestandsnaam in te stellen het loggen multithreaded zou kunnen zijn, wat de uitvoer in hoge mate vertraagt.

HOOFDSTUK 14

Infrastructuur voor authenticatie

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

```
from qgis.core import (
  QgsApplication,
  QgsRasterLayer,
  QgsAuthMethodConfig,
  QgsDataSourceUri,
  QgsPkiBundle,
  QgsMessageLog,
)

from qgis.gui import (
    QgsAuthAuthoritiesEditor,
    QgsAuthConfigEditor,
    QgsAuthConfigSelect,
    QgsAuthSettingsWidget,
)

from qgis.PyQt.QtWidgets import (
    QWidget,
    QTabWidget,
)

from qgis.PyQt.QtNetwork import QSslCertificate
```

14.1 Introductie

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

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

Het systeem voor authenticatie wordt in QGIS Desktop breed gebruikt door gegevensproviders als inloggegevens worden vereist om toegang te krijgen tot een bepaalde bron, bijvoorbeeld wanneer een laag een verbinding maakt met een database voor Postgres.

Er zijn ook een paar widgets in de bibliotheek voor de GUI van QGIS die ontwikkelaars voor plug-ins kunnen gebruiken om de infrastructuur voor de authenticatie gemakkelijk te integreren in hun code:

- `QgsAuthConfigEditor`
- `QgsAuthConfigSelect`
- `QgsAuthSettingsWidget`

Een goede verwijzing voor de code is te lezen in de infrastructuur voor de authenticatie tests code.

Waarschuwing: Vanwege de beperkingen voor de beveiliging, waarmee rekening werd gehouden bij het ontwerpen van de infrastructuur voor de authenticatie, wordt alleen een geselecteerde subset van de interne methoden weergegeven in Python.

14.2 Woordenlijst

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

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

Authenticatie-database
: 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.

14.3 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.

14.3.1 Initialiseren van de beheerder en het hoofdwachtwoord instellen

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

```
authMgr = QgsApplication.authManager()

# check if QgsAuthManager has already been 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.authenticationDatabasePath():
    # already initialized => 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 checks if the hash of the password was
        # already saved in the authentication db
        assert authMgr.setMasterPassword("your master password",
                                         verify=True), msg
else:
    # outside qgis, e.g. in a testing environment => setup env var before
    # db init
    os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
    msg = 'Master password could not be set'
    assert authMgr.setMasterPassword("your master password", True), msg
    authMgr.init("/path/where/located/qgis-auth.db")
```

14.3.2 Vullen van authdb met een nieuw item Configuratie voor authenticatie

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 tekenreeks wordt automatisch gegenereerd bij het maken van een item met de QGIS API of GUI, maar het zou nuttig kunnen zijn om het handmatig in te stellen met een bekende waarde in het geval dat de configuratie moet worden gedeeld (met verschillende inloggegevens) tussen meerdere gebruikers binnen een organisatie.

`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 = QgsApplication.authManager()
# set alice PKI data
config = QgsAuthMethodConfig()
config.setName("alice")
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
config.setMethod("PKI-Paths")
config.setUri("https://example.com")
config.setConfig("certpath", "path/to/alice-cert.pem" )
config.setConfig("keypath", "path/to/alice-key.pem" )
# check if method parameters are correctly set
assert config.isValid()

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

Beschikbare authenticatiemethoden

bibliotheken *Authentication Method* worden dynamisch geladen gedurende de initialisatie van de beheerder voor de authenticatie. Beschikbare methoden voor authenticatie zijn:

1. `Basic` Gebruiker en wachtwoordauthenticatie
2. `EsriToken` Op ESRI-token gebaseerde authenticatie
3. `Identity-Cert` Authenticatie met Identiteitscertificaat
4. `OAuth2` OAuth2-authenticatie
5. `PKI-Paths` Authenticatie met PKI-paden
6. `PKI-PKCS#12` Authenticatie met PKI PKCS#12

Autoriteiten vullen

```
authMgr = QgsApplication.authManager()
# 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()
```

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
caBundlesList = []  # List of CA bundles
bundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                    "/path/to/alice-key_w-pass.pem",
                                    "unlock_pwd",
                                    caBundlesList )
assert bundle is not None
# You can check bundle validity by calling:
# bundle.isValid()
```

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

14.3.3 Een item verwijderen uit authdb

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

```
authMgr = QgsApplication.authManager()
authMgr.removeAuthenticationConfig( "authCfg_Id_to_remove" )
```

14.3.4 Uitbreiden van authcfg overlaten aan QgsAuthManager

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

Notitie: 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:

```
authM = QgsApplication.authManager()
print(authM.authMethod("Identity-Cert").supportedDataProviders())
```

Voorbeeld uitvoer:

```
['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(str(quri.encodedUri(), "utf-8"), '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:

```
authCfg = 'fm1s770'
quri = QgsDataSourceUri("my WMS uri here")
quri.setParam("authcfg", authCfg)
rlayer = QgsRasterLayer( quri.uri(False), 'states', 'wms')
```

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

PKI-voorbeelden met andere gegevensproviders

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

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

Vele plug-ins van derde partijen gebruiken httplib2 of andere bibliotheken van Python voor netwerken om verbindingen van HTTP te beheren, in plaats van ze te integreren met `QgsNetworkAccessManager` en zijn gerelateerde integratie van de Authentication Infrastructure .

Een hulpfunctie voor Python is gemaakt om deze integratie te faciliteren, genaamd `NetworkAccessManager`. De code ervan is hier te vinden.

Deze hulpklasse kan worden gebruikt zoals in het volgende snippet:

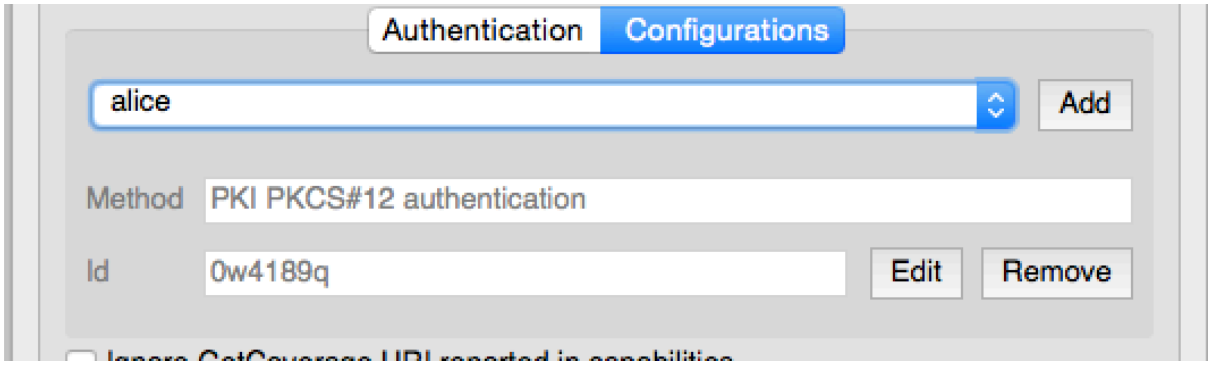
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 GUI's voor authenticatie

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

14.5.1 GUI om persoonlijke gegevens te selecteren

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



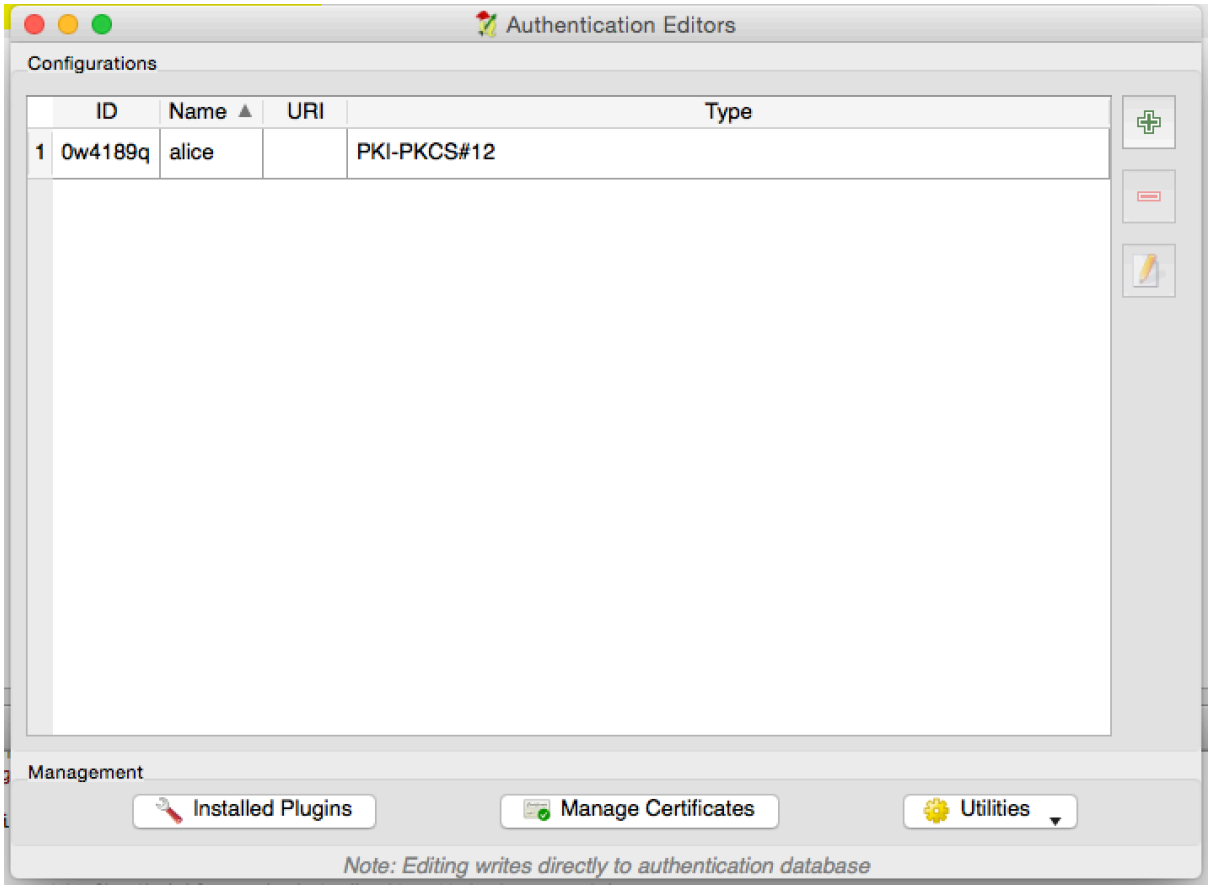
en kan worden gebruikt zoals in het volgende snippet:

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

Het bovenstaande voorbeeld is afkomstig uit de broncode van QGIS code. De tweede parameter van de gebruikersinterface verwijst naar het type gegevensprovider. De parameter wordt gebruikt om de compatibele *Authentication Method* te beperken tot de gespecificeerde provider.

14.5.2 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`.



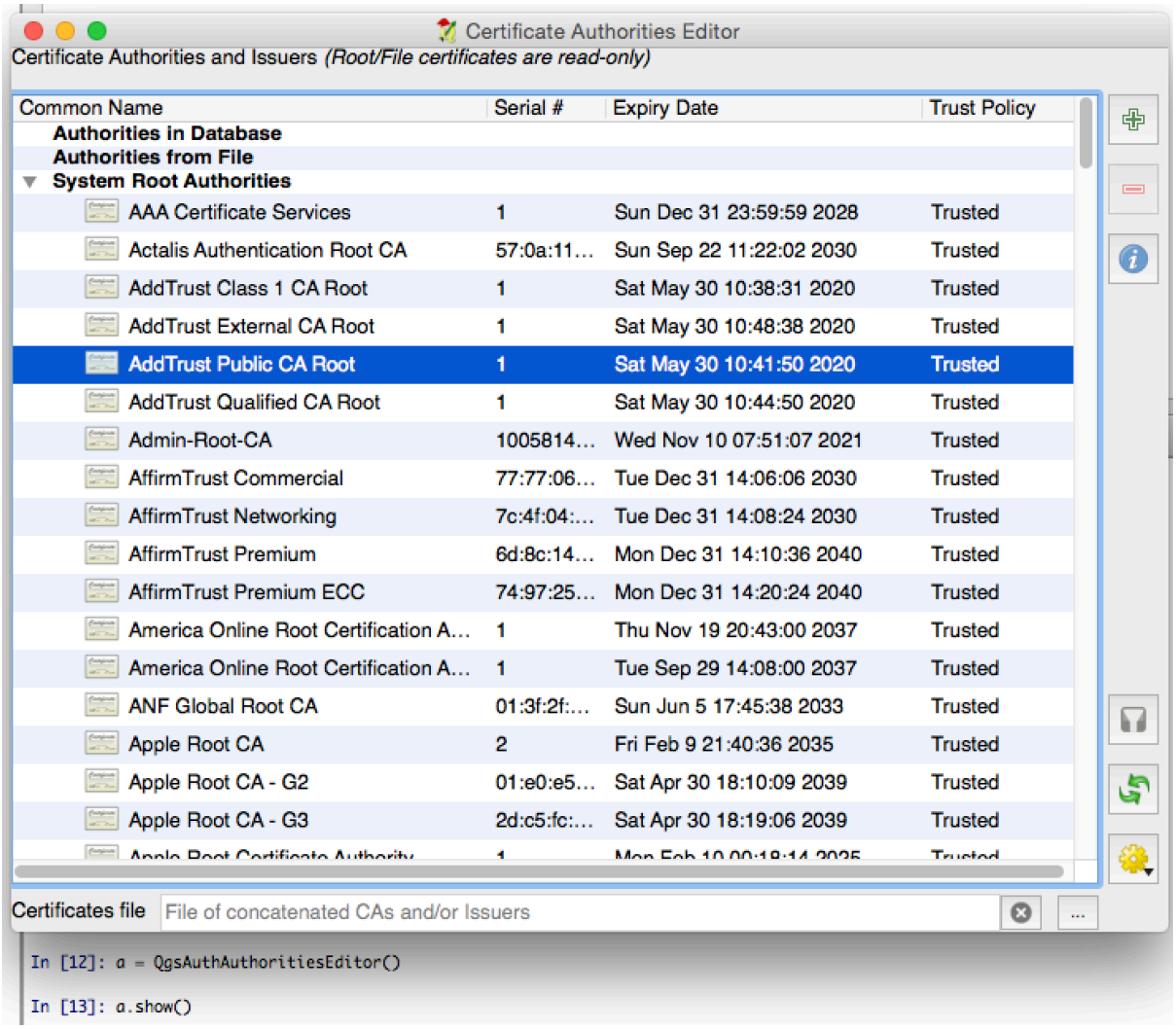
en kan worden gebruikt zoals in het volgende snippet:

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

Een geïntegreerd voorbeeld is te vinden in de gerelateerde test.

14.5.3 Bewerker voor GUI autoriteiten

Een GUI die alleen wordt gebruikt voor het beheren van autoriteiten wordt beheerd door de klasse `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`
parent = QWidget()  # Your GUI parent widget
gui = QgsAuthAuthoritiesEditor( parent )
gui.show()
```

HOOFDSTUK 15

Taken - veel werk op de achtergrond doen

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

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

15.1 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 worden 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):
    pass
```

- Maak een taak uit een functie.

```
def heavyFunction():
    # Some CPU intensive processing ...
    pass

def workdone():
    # ... do something useful with the results
    pass

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

- Maak een taak uit een algoritme van Processing.

```
params = dict()
context = QgsProcessingContext()
context.setProject(QgsProject.instance())
feedback = QgsProcessingFeedback()

buffer_alg = QgsApplication.instance().processingRegistry().algorithmById(
↪'native:buffer')
task = QgsProcessingAlgRunnerTask(buffer_alg, params, context,
                            feedback)
```

Waarschuwing: Elke taak op de achtergrond (ongeacht hoe die is gemaakt) moet NOOIT een QObject gebruiken dat leeft in de hoofdthread, zoals toegang tot QgsVectorLayer, QgsProject of 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. Gegevens die worden gebruikt in een taak moeten zijn gekopieerd voordat de taak wordt gestart. Pogingen om ze te gebruiken vanuit threads voor de achtergrond zal leiden tot crashes.

Meerr nog, zorg er altijd voor dat `context` en `feedback` levend zijn voor ten minste zo lang als de taken duren die ze gebruiken. QGIS zal crashen als, bij het voltooien van een taak, *QgsTaskManager* faalt bij het verkrijgen van toegang tot de *context* en *feedback* voor de geplande taken.

Notitie: Het is een veel voorkomend patroon om `setProject()` aan te roepen, kort nadat `QgsProcessingContext` is aangeroepen. Dat maakt het voor de taak, en ook zijn functie callback, mogelijk om de meeste project-brede instellingen te gebruiken. Dit is in het bijzonder van waarde bij het werken met ruimtelijke lagen in de functie callback.

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`.

15.2 Voorbeelden

15.2.1 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, Qgis
    )

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
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        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(
                'RandomTask "{name}" completed\n' \
                'RandomTotal: {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(
                    'RandomTask "{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(
                    'RandomTask "{name}" Exception: {exception}'.format(
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception

    def cancel(self):
        QgsMessageLog.logMessage(
            'RandomTask "{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)
```

```
RandomIntegerSumTask(0): Started task "waste cpu subtask shortest"
RandomIntegerSumTask(0): Started task "waste cpu short"
RandomIntegerSumTask(0): Started task "waste cpu mini"
RandomIntegerSumTask(0): Started task "waste cpu subtask long"
RandomIntegerSumTask(3): Task "waste cpu subtask shortest" completed
RandomTotal: 25452 (with 100 iterations)
RandomIntegerSumTask(3): Task "waste cpu mini" completed
RandomTotal: 23810 (with 100 iterations)
RandomIntegerSumTask(3): Task "waste cpu subtask long" completed
RandomTotal: 26308 (with 100 iterations)
RandomIntegerSumTask(0): Started task "waste cpu long"
RandomIntegerSumTask(3): Task "waste cpu long" completed
RandomTotal: 22534 (with 100 iterations)
```

15.2.2 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()),
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        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

# Create 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)
```

```
RandomIntegerSumTask(0): Started task "waste cpu subtask short"
RandomTaskFromFunction(0): Started task Waste cpu 1
RandomTaskFromFunction(0): Started task Waste cpu 2
RandomTaskFromFunction(0): Task Waste cpu 2 completed
RandomTotal: 23263 ( with 100 iterations)
RandomTaskFromFunction(0): Task Waste cpu 1 completed
RandomTotal: 25044 ( with 100 iterations)
```

15.2.3 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)
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    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` and `feedback` need to
# live for as least as long as `task`,
# otherwise the program will crash.
# Initializing them globally is a sure way
# of avoiding this unfortunate situation.
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)
```

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

HOOFDSTUK 16

Python plug-ins ontwikkelen

16.1 Plug-ins voor Python structureren

De belangrijkste stappen voor het maken van een plug-in zijn:

1. *Idee*: Heb een idee over wat u wilt doen met uw nieuwe plug-in voor QGIS.
2. *Instellen*: *De bestanden voor uw plug-in maken*. Afhankelijk van het type plug-in, sommige zijn verplicht, waar andere optioneel zijn
3. *Ontwikkel*: *De code schrijven* in toepasselijke bestanden
4. *Documenteren*: *De documentatie voor de plug-in schrijven*
5. Optioneel: *Vertalen*: *Uw plug-in vertalen* in verschillende talen
6. *Testen*: *Uw plug-in opnieuw laden* om te controleren of alles OK is
7. *Publiceren*: Publiceer uw plug-in in de opslagplaats van QGIS of maak uw eigen opslagplaats als een “arsenaal” van persoonlijke “wapens voor GIS”.

16.1.1 Beginnen

Kijk eerst eens in de *Officiële Python plug-in opslagplaats* vóór het beginnen te schrijven van een nieuwe plug-in. De broncode van bestaande plug-ins kan u helpen meer te leren over programmeren. U zou ook kunnen vinden dat een soortgelijke plug-in al bestaat en u zou in staat kunnen zijn die uit te breiden of tenminste erop doorbouwen om uw eigen te ontwikkelen.

De bestandsstructuur van de plug-in instellen

We moeten de noodzakelijk bestanden voor de plug-in instellen om te beginnen met een nieuwe plug-in.

Er zijn twee sjabloonbronnen voor plug-ins die u op weg zouden kunnen helkpen:

- Voor educatieve doeleinden of wanneer een minimalistische benadering gewenst is, verschaft de minimal plugin template de noodzakelijke basisbestanden (skelet) om een geldige plug-in voor Python in QGIS te maken.
- Voor een meer volledige sjabloon voor plug-ins kan de Plugin Builder sjablonen maken voor meerdere verschillende typen plug-ins, inclusief objecten mogelijkheden, zoals lokalisatie (vertaling) en testen.

Een map voor een normale plug-in bevat de volgende bestanden:

- `metadata.txt` - *vereist* - Bevat algemene informatie, versie, naam en enkele andere metadata, gebruikt door de website van de plug-in en infrastructuur van de plug-in.
- `__init__.py` - *vereist* - Het beginpunt van de plug-in. Het moet de methode `classFactory()` hebben en mag elke andere code voor initialisatie hebben.
- `mainPlugin.py` - *broncode* - De belangrijkste werkende code van de plug-in. Bevat alle informatie over de acties van de plug-in en de hoofdcode.
- `form.ui` - *voor plug-ins met aangepaste gebruikersinterface* - De gebruikersinterface die is gemaakt met Qt Designer.
- `form.py` - *gecompileerde gebruikersinterface* - De vertaling van form.ui, hierboven beschreven, naar Python.
- `resources.qrc` - *optioneel* - Het door Qt Designer gemaakte .xml-document. Bevat relatieve paden naar de bronnen van de formulieren voor de gebruikersinterface.
- `resources.py` - *gecompileerde bronnen, optioneel* - De vertaling van het bestand .qrc, hierboven beschreven, naar Python.
- `LICENSE` - *vereist* als de plug-in moet worden gepubliceerd of bijgewerkt in de QGIS Plugins Directory, anders *optioneel*. Bestand zou een platte tekst-bestand moeten zijn zonder bestandsextensie in de bestandsnaam.

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*.

16.1.2 Een plug-in schrijven

Het volgende gedeelte laat zien welke inhoud zou moeten worden ingevoegd in elke van de hierboven geïntroduceerde bestanden.

metadata.txt

Als eerste moet Plug-ins beheren en installeren enige basisinformatie ophalen over de plug-in, zoals de naam, omschrijving etc. ervan. Deze informatie wordt opgeslagen in `metadata.txt`.

Notitie: Alle metadata moet in de codering UTF-8 zijn.

Naam van de metadata	Verei	Opmerkingen
name	Ja	een korte string die de naam van de plug-in bevat
qgisMinimum	Ja	notatie, met punten, van de minimale versie van QGIS
qgisMaximum	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` - *True* als deze versie experimenteel is
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`, `Mesh` en `Web`
plugin_depend	Nee	PIP-achtige kommagescheiden lijst van andere te installeren plug-ins, gebruik de namen van de plug-in vanuit hun veld voor de naam in de metadata
server	Nee	Booleaanse vlag, `True` of `False`, bepaalt of de plug-in een server-interface heeft
hasProcessingP	Nee	Booleaanse vlag, `True` of `False`, bepaalt of de plug-in algoritmes voor Processing verschaft

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

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    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

; Since QGIS 3.8, a comma separated list of plugins to be installed
; (or upgraded) can be specified.
; The example below will try to install (or upgrade) "MyOtherPlugin" version 1.12
; and any version of "YetAnotherPlugin".
; Both "MyOtherPlugin" and "YetAnotherPlugin" names come from their own metadata's
; name field
plugin_dependencies=MyOtherPlugin==1.12,YetAnotherPlugin
```

__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("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__` dat toegang geeft tot de interface van QGIS
- `initGui()` aangeroepen als de plug-in wordt geladen
- `unload()` aangeroepen als 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("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.

Hoewel acties Help en Over ook kunnen worden toegevoegd aan uw aangepaste menu, is een handiger plaats om ze beschikbaar te maken het QGIS hoofdmenu van *Help* ► *Plug-ins*. Dit wordt gedaan met de methode `pluginHelpMenu()`.

```
def initGui(self):

    self.help_action = QAction(
        QIcon("testplug:icon.png"),
        self.tr("Test Plugin..."),
        self.iface.mainWindow()
    )
    # Add the action to the Help menu
    self.iface.pluginHelpMenu().addAction(self.help_action)

    self.help_action.triggered.connect(self.show_help)

@staticmethod
def show_help():
    """ Open the online help. """
    QDesktopServices.openUrl(QUrl('https://docs.qgis.org'))

def unload(self):

    self.iface.pluginHelpMenu().removeAction(self.help_action)
    del self.help_action
```

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.

16.1.3 Documenteren van plug-ins

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.

16.1.4 Vertalen van plug-ins

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, om de vertaling van uw plug-in te kunnen zien, QGIS, wijzig de taal (*Extra* ► *Opties* ► *Algemeen*) en start QGIS opnieuw.

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.

16.1.5 Delen van uw plug-in

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.

16.1.6 Tips en trucs

Plugin Reloader

Gedurende de ontwikkeling van uw plug-in zult u die regelmatig opnieuw in QGIS moeten laden om te testen. Dat is heel gemakkelijk met de plug-in **Plugin Reloader**. Die kunt u vinden met Plug-ins beheren en installeren.

Automatiseren van verpakken, uitgave en vertaling met qgis-plugin-ci

qgis-plugin-ci verschaft een interface voor de opdrachtregel om automatisch verpakken en uitrollen voor plug-ins van QGIS op uw computer uit te voeren, of doorlopende integratie te gebruiken, zoals GitHub werkstromen of Gitlab-CI als ook Transifex voor vertaling.

Het maakt uitgeven, vertalen, publiceren of maken van een XML-bestand voor de opslagplaats van plug-ins via CLI of in acties voor CI mogelijk.

Toegang tot plug-ins

Met Python kunt u toegang krijgen tot alle klassen van geïnstalleerde plug-ins vanuit QGIS, wat handig kan zijn voor debuggen.

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

Logboekmeldingen

Plug-ins hebben hun eigen tab in het log_message_panel.

Bronbestand

Sommige plug-ins gebruiken resource-bestanden, bijvoorbeeld `resources.qrc` dat de bronnen voor de gebruikersinterface definieert, zoals pictogrammen:

```
<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 de poging voor het uitvoeren van de opdracht **pyrcc5**, vanuit de Opdrachtprompt 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`.

16.2 Codesnippers

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

```
from qgis.core import (
    QgsProject,
    QgsApplication,
    QgsMapLayer,
)

from qgis.gui import (
    QgsGui,
    QgsOptionsWidgetFactory,
    QgsOptionsPageWidget,
    QgsLayerTreeEmbeddedWidgetProvider,
    QgsLayerTreeEmbeddedWidgetRegistry,
)

from qgis.PyQt.QtCore import Qt
from qgis.PyQt.QtWidgets import (
    QMessageBox,
    QAction,
    QHBoxLayout,
    QComboBox,
)
from qgis.PyQt.QtGui import QIcon
```

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

16.2.1 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")
```

Het is ook mogelijk om gebruikers toe te staan sneltoetsen voor het toetsenbord te laten aanpassen voor de verschafte acties. Dit wordt gedaan door toe te voegen:

```
# in the initGui() function
QgsGui.shortcutsManager().registerAction(self.key_action)

# and in the unload() function
QgsGui.shortcutsManager().unregisterAction(self.key_action)
```

16.2.2 Hoe pictogrammen van QGIS opnieuw te gebruiken

Omdat zij goed bekend zijn en een heldere boodschap overbrengen naar de gebruikers, wilt u misschien soms pictogrammen van QGIS opnieuw in uw plug-in gebruiken in plaats van een nieuwe te tekenen en in te stellen. Gebruik de methode `getThemeIcon()`.

Bijvoorbeeld om het pictogram  mActionFileOpen.svg, beschikbaar in de QGIS opslagplaats voor code, opnieuw te gebruiken:

```
# e.g. somewhere in the initGui
self.file_open_action = QAction(
    QgsApplication.getThemeIcon("/mActionFileOpen.svg"),
    self.tr("Select a File..."),
    self.iface.mainWindow()
)
self.iface.addPluginToMenu("MyPlugin", self.file_open_action)
```

`iconPath()` is een andere methode om pictogrammen van QGIS aan te roepen. Voorbeelden van het aanroepen van thema-pictogrammen zijn te vinden op QGIS embedded images - Cheatsheet.

16.2.3 Interface voor plug-in in het dialoogvenster Opties

U kunt een aangepaste tab voor plug-ins Opties toevoegen aan *Extra* ► *Opties*. Dit heeft de voorkeur boven het toevoegen van een specifiek item voor het hoofdmenu voor opties van uw plug-in, omdat het alle instellingen voor de toepassing QGIS behoudt en instellingen voor de plug-in op één enkele plaats staan, wat gemakkelijk is voor gebruikers om te ontdekken en te navigeren.

De volgende snipper zal slechts een blanco tab voor de instellingen voor de plug-in toevoegen, die u kunt vullen met alle opties en instellingen, specifiek voor uw plug-in. U kunt de volgende klassen splitsen in verschillende bestanden. In dit voorbeeld voegen we twee klassen toe aan het hoofdbestand `mainPlugin.py`.

```
class MyPluginOptionsFactory(QgsOptionsWidgetFactory):

    def __init__(self):
        super().__init__()

    def icon(self):
        return QIcon('icons/my_plugin_icon.svg')

    def createWidget(self, parent):
        return ConfigOptionsPage(parent)


class ConfigOptionsPage(QgsOptionsPageWidget):

    def __init__(self, parent):
        super().__init__(parent)
        layout = QHBoxLayout()
        layout.setContentsMargins(0, 0, 0, 0)
        self.setLayout(layout)
```

Tenslotte voegen we de imports toe en passen de functie `__init__` aan:

```
from qgis.PyQt.QtWidgets import QHBoxLayout
from qgis.gui import QgsOptionsWidgetFactory, QgsOptionsPageWidget


class MyPlugin:
    """QGIS Plugin Implementation."""

    def __init__(self, iface):
        """Constructor.

        :param iface: An interface instance that will be passed to this class
            which provides the hook by which you can manipulate the QGIS
            application at run time.
        :type iface: QgsInterface
        """
        # Save reference to the QGIS interface
        self.iface = iface


    def initGui(self):
        self.options_factory = MyPluginOptionsFactory()
        self.options_factory.setTitle(self.tr('My Plugin'))
        iface.registerOptionsWidgetFactory(self.options_factory)

    def unload(self):
        iface.unregisterOptionsWidgetFactory(self.options_factory)
```

Tip: Aangepaste tabs toevoegen aan een dialoogvenster voor laag-eigenschappen

U kunt een soortgelijke logica toepassen om de aangepaste optie voor de plug-in toe te voegen aan het dialoogvenster voor de laageigenschappen met de klassen `QgsMapLayerConfigWidgetFactory` en `QgsMapLayerConfigWidget`.

16.2.4 Aangepaste widgets voor lagen inbedden in de boom van lagen

Naast de normale elementen voor symbologie van lagen, weergegeven naast of onder het item van de laag in het paneel *Lagen*, kunt u uw eigen widgets toevoegen, wat het mogelijk maakt snelle toegang te enkele acties die vaak gebruikt worden met een laag (instellen van filteren, selectie, stijl, vernieuwen van een laag met een widget voor een knop, een op een laag gebaseerd schuifbalk voor tijd maken of eenvoudigweg extra informatie over de laag weergeven in een Label daar, of …). Deze zogenaamde **In boom van lagen ingebedde widgets** worden voor individuele lagen beschikbaar gemaakt op de tab *Legenda* van de laageigenschappen.

De volgende codesnipper maakt een keuzemenu in de legenda die u de beschikbare laagstijlen voor de laag laat zien, wat het mogelijk maakt snel te schakelen tussen de verschillende laagstijlen.

```
class LayerStyleComboBox(QComboBox):
    def __init__(self, layer):
        QComboBox.__init__(self)
        self.layer = layer
        for style_name in layer.styleManager().styles():
            self.addItem(style_name)

        idx = self.findText(layer.styleManager().currentStyle())
        if idx != -1:
          self.setCurrentIndex(idx)

        self.currentIndexChanged.connect(self.on_current_changed)

    def on_current_changed(self, index):
        self.layer.styleManager().setCurrentStyle(self.itemText(index))

class LayerStyleWidgetProvider(QgsLayerTreeEmbeddedWidgetProvider):
    def __init__(self):
        QgsLayerTreeEmbeddedWidgetProvider.__init__(self)

    def id(self):
        return "style"

    def name(self):
        return "Layer style chooser"

    def createWidget(self, layer, widgetIndex):
        return LayerStyleComboBox(layer)

    def supportsLayer(self, layer):
        return True   # any layer is fine

provider = LayerStyleWidgetProvider()
QgsGui.layerTreeEmbeddedWidgetRegistry().addProvider(provider)
```

Sleep dan, vanaf de tab met eigenschappen voor de *Legenda* voor een bepaalde laag, de `Laagstijl kiezen` vanuit de *Beschikbare widgets* naar *Gebruikte widgets* om de widget in de boom met lagen in te schakelen. Ingebedde worden ALTIJD weergegeven aan de bovenzijde van hun geassocieerde sub-item van de knoop van de laag.

Als u de widgets wilt gebruiken vanuit bijvoorbeeld een plug-in, kunt u ze als volgt toevoegen:

```
layer = iface.activeLayer()
counter = int(layer.customProperty("embeddedWidgets/count", 0))
layer.setCustomProperty("embeddedWidgets/count", counter+1)
layer.setCustomProperty("embeddedWidgets/{}/id".format(counter), "style")
view = self.iface.layerTreeView()
view.layerTreeModel().refreshLayerLegend(view.currentLegendNode())
view.currentNode().setExpanded(True)
```

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

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.

16.3.1 Nuttige plug-ins voor het schrijven van plug-ins in Python

Sommige plug-ins zijn handig bij het schrijven van plug-ins in Python. Installeer vanuit *Plug-ins* ► *Plug-ins beheren en installeren…*:

- *Plugin reloader*: Dit laat u een plug-in opnieuw laden en nieuwe wijzigingen op te halen, zonder QGIS opnieuw te starten.
- *First Aid*: Dit zal een Python-console en lokale debugger toevoegen om variabelen te inspecteren als een uitzondering wordt opgeworpen vanuit een plug-in.

Waarschuwing: Ondanks onze constante inspanningen zou informatie beneden deze regel nog niet kunnen zijn bijgewerkt naar QGIS 3.

16.3.2 Een opmerking bij het configureren van uw IDE op Linux en Windows

Op Linux, alles wat gewoonlijk moet worden gedaan is de locaties van de bibliotheken van QGIS toe te voegen aan de omgevingsvariabele `PYTHONPATH` van de gebruiker. In de meeste distributies kan dit worden gedaan door `~/.bashrc` of `~/.bash-profile` te bewerken met de volgende regel (getest op OpenSUSE Tumbleweed):

```
export PYTHONPATH="$PYTHONPATH:/usr/share/qgis/python/plugins:/usr/share/qgis/
↪python"
```

Sla het bestand op en implementeer de instelling voor de omgeving door de volgende opdracht voor de shell te gebruiken:

```
source ~/.bashrc
```

Op Windows dient u er voor te zorgen dat u dezelfde instellingen voor de omgeving hebt en dezelfde bibliotheken en interpreter gebruikt als QGIS. De snelste manier om dit te doen is om het batchbestand voor het opstarten 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`.

16.3.3 Debuggen met behulp van Pyscripter IDE (Windows)

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

1. Maak een kopie van `qgis-unstable.bat` en hernoem die naar `pyscripter.bat`.
2. Open het in een bewerker. En verwijder de laatste regel, die welke QGIS laat starten.
3. 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
4. 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
```

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

Populairder dan Pyscripter is Eclipse een veelgebruikte keuze voor ontwikkelaars. In het volgende gedeelte zullen we uitleggen hoe het te configureren voor het ontwikkelen en testen van plug-ins.

16.3.4 Debuggen met behulp van Eclipse en PyDev

Installeren

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

- Eclipse
- Aptana Studio 3 Plugin of PyDev
- QGIS 3.x
- U zou misschien ook **Remote Debug** willen installeren, een plug-in voor QGIS. Op dit moment is die nog steeds experimenteel, dus schakel eerst ☑ *Ook de experimentele plug-ins tonen* onder *Plug-ins* ► *Plug-ins beheren en installeren…* ► *Extra* in.

U zou ook een batchbestand moeten maken en het gebruiken om Eclipse te starten, om uw omgeving voor te bereiden voor het gebruiken van Eclipse in Windows:

1. Zoek naar de map waar het bestand `qgis_core.dll` is geplaatst. Normaal gesproken is dit `C:\OSGeo4W\apps\qgis\bin`, maar als u uw eigen toepassing in QGIS compileerde staat het in de map waar u het bouwde in `output/bin/RelWithDebInfo`
2. Zoek naar uw uitvoerbare bestand `eclipse.exe`.
3. 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
start /B C:\path\to\your\eclipse.exe
```

Eclipse instellen

1. 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.

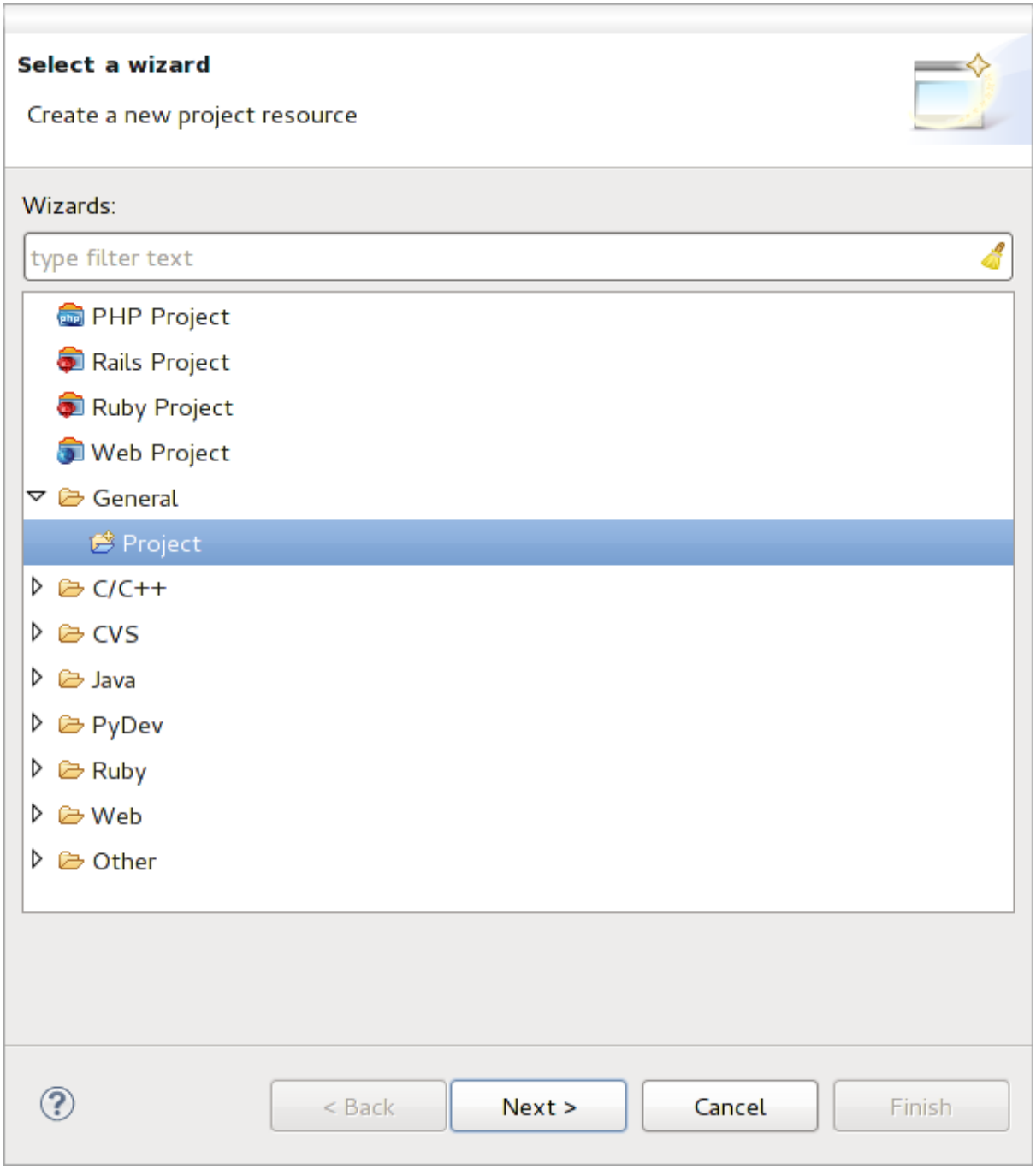


Fig. 16.1: Eclipse-project

2. Klik met rechts op uw nieuwe project en kies *New* ► *Folder*.
3. 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

De debugger werkend maken:

1. Schakel naar het Perspective Debug in Eclipse (*Window* ► *Open Perspective* ► *Other* ► *Debug*).
2. start de PyDev debug server door te kiezen *PyDev* ► *Start Debug Server*.
3. 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 )
```

Fig. 16.2: 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.

1. Open de Console view (*Window* ► *Show view*). Het zal de console van *Debug Server* weergeven, wat niet erg interessant is. Maar er is een knop *Open Console* die u laat schakelen naar een meer interessante PyDev Debug Console.
2. 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.
3. Kies *PyDev Debug Console*. In het geval het uitgegrijsd is en het u vertelt de debugger te starten en het geldige frame te selecteren, zorg er voor dat u de debugger op afstand hebt en momenteel op een onderbrekingspunt staat.

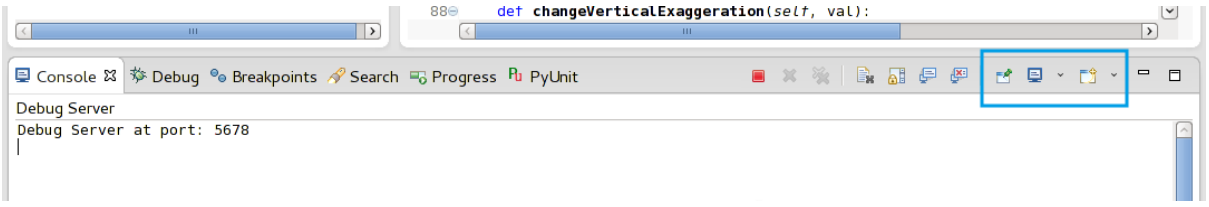


Fig. 16.3: PyDev console voor debuggen

U hebt nu een interactieve console die u alle opdrachten laat testen vanuit de de huidige context. U kunt variabelen bewerken of aanroepen naar de API maken of wat u maar wilt.

Tip: 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.

1. 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*.

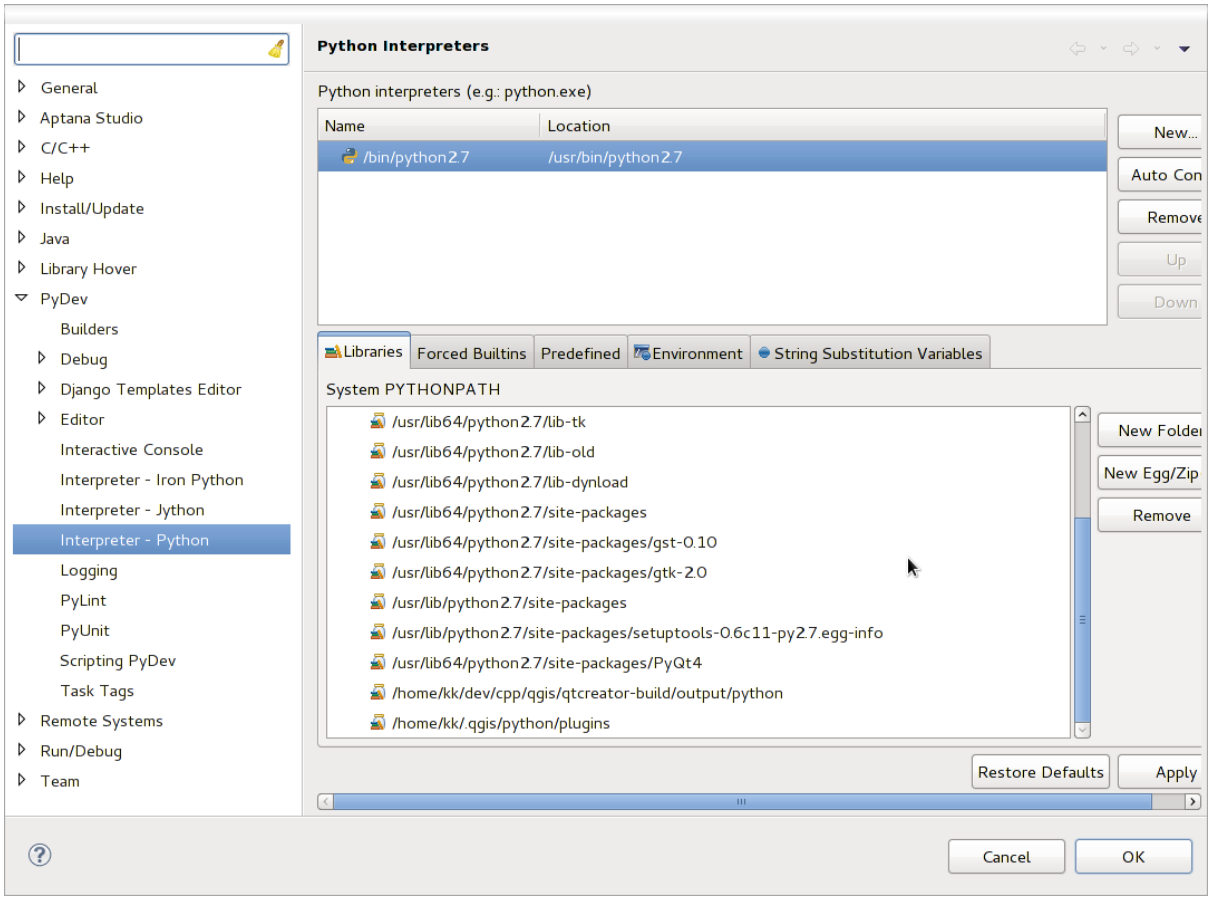


Fig. 16.4: PyDev console voor debuggen

2. Open eerst de tab Libraries.
3. Voeg een New Folder toe en kies de map voor Python van uw installatie van QGIS. Als u niet weet waar deze map staat (het is niet de map plugins):
   1. Open QGIS
   2. Start een console voor Python

3. Voer in `qgis`
4. en druk op Enter. Het zal u laten zien welke module van QGIS het gebruikt en het pad ervan.
5. Verwijder het achterliggende gedeelte `/qgis/__init__.pyc` uit dit pad en u heeft het pad waar u naar zoekt.

4. U zou hier ook uw map voor plug-ins moeten toevoegen (het staat in `python/plugins` in de map gebruikersprofiel).
5. 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 PyQt. Voeg daarom ook PyQt toe als forced builtin. Die zou waarschijnlijk al aanwezig zijn op uw tab Libraries.
6. Klik op *OK* en u bent klaar.

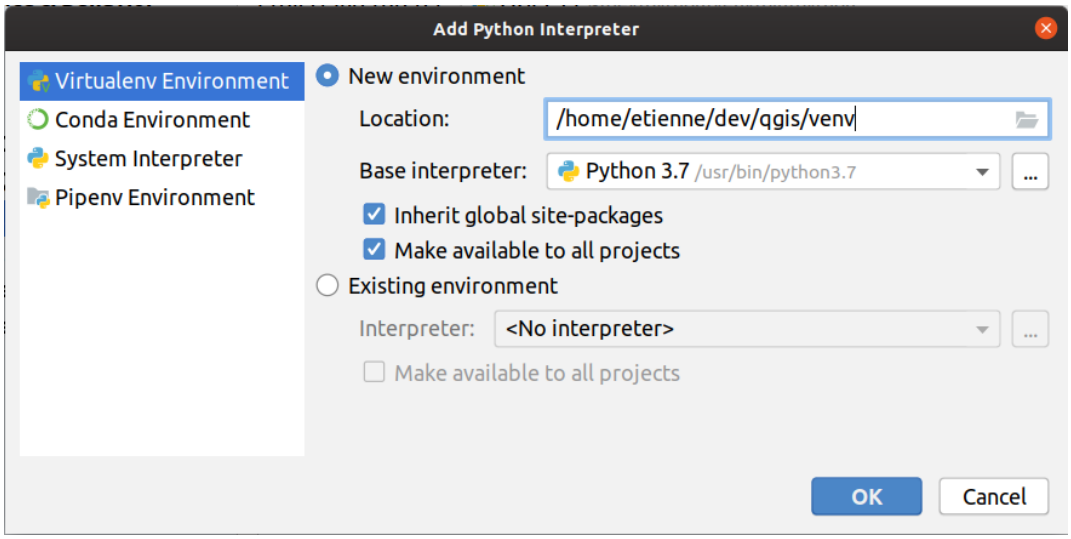
Notitie: Elke keer dat de API van QGIS wijzigt (bijv. als u de master van QGIS compileert en het bestand SIP wijzigt), zou u terug moeten gaan naar deze pagina en eenvoudigweg op *Apply* moeten klikken. Dat laat Eclipse alle bibliotheken opnieuw parsen.

16.3.5 Debuggen met PyCharm op Ubuntu met een gecompileerde QGIS

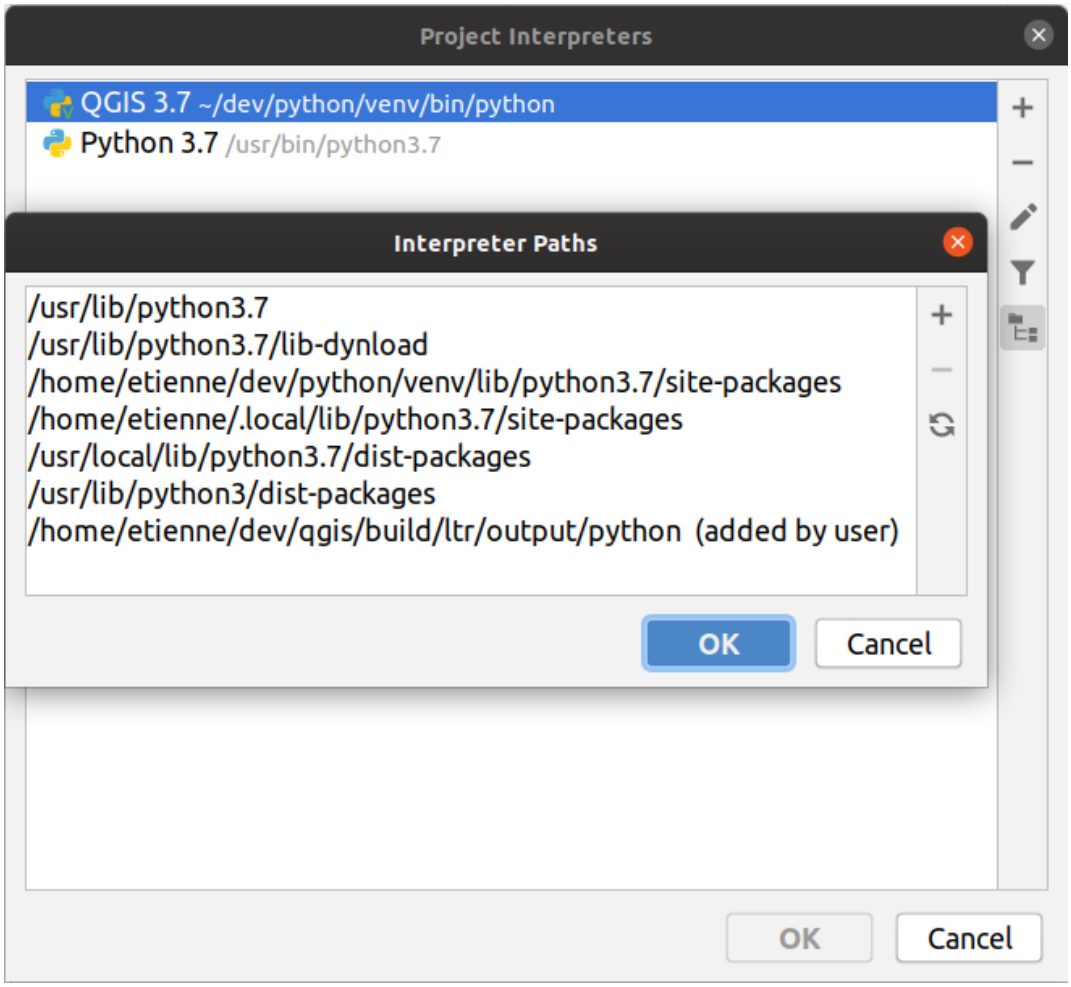
PyCharm is een IDE voor Python, ontwikkeld door JetBrains. Er is een gratis versie, genaamd Community Edition, en een waarvoor betaald moet worden, genaamd Professional. U kunt PyCharm downloaden vanaf de website: https://www.jetbrains.com/pycharm/download

We nemen aan dat u QGIS hebt gecompileerd op Ubuntu met de opgegeven map voor het bouwen `~/dev/qgis/build/master`. Het is niet verplicht een eigen gecompileerde QGIS te hebben, maar alleen dit is getest. Paden moeten worden aangepast.

1. In PyCharm, in uw *Project Properties*, *Project Interpreter*, gaan we een Python Virtual environment maken, genaamd `QGIS`.
2. Klik op het kleine tandwiel en dan op *Add*.
3. Selecteer *Virtualenv environment*.
4. Selecteer een algemene locatie voor al uw projecten van Python, zoals `~/dev/qgis/venv`, omdat we deze interpreter voor Python zullen gebruiken voor al onze plug-ins.
5. Kies een op uw systeem beschikbare interpreter op basis van Python 3 en selecteer de volgende twee opties *Inherit global site-packages* en *Make available to all projects*.



1. Klik op *OK*, ga terug naar het kleine tandwiel en klik op *Show all*.
2. Selecteer, in het nieuwe venster, uw nieuwe interpreter `QGIS` en klik op het laatste pictogram in het verticale menu *Show paths for the selected interpreter*.
3. Voeg tenslotte het volgende absolute pad toe aan de lijst `~/dev/qgis/build/master/output/python`.



1. Start PyCharm opnieuw op en u kunt beginnen met het gebruiken van deze nieuwe virtuele omgeving van Python voor al uw plug-ins.

PyCharm zal zich bewust zijn van de API van QGIS en ook van de API van PyQt, als u Qt gebruikt zoals het wordt verschaft door QGIS,zoals `from qgis.PyQt.QtCore import QDir`. Het automatisch aanvullen zou moeten werken en PyCharm kan uw code inspecteren.

In de professionele versie van PyCharm, werkt debuggen op afstand heel goed. Voor de Community edition is debuggen op afstand niet beschikbaar. U hebt alleen toegang tot een lokale debugger, wat betekent dat de code moet worden uitgevoerd *binnen* PyCharm (als script of eenheidstest), niet in QGIS zelf. Voor Pythoncode die wordt uitgevoerd *in* QGIS, zou u de plug-in *First Aid*, zoals hierboven vermeld, kunnen gebruiken.

16.3.6 Debuggen met behulp van PDB

Wanneer u geen IDE gebruikt, zoals Eclipse of PyCharm, kunt u debuggen met PDB, door deze stappen te volgen.

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

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

2. Voer dan QGIS uit vanaf de opdrachtregel.

   Doe op Linux:

```
$ ./Qgis
```

   Doe op macOS:

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

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

TODO:
Informatie voor testen toevoegen

16.4 Uw plug-in uitgeven

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:

16.4.1 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

16.4.2 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.

16.4.3 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
5. een licentiebestand moet zijn opgenomen, opgeslagen als `LICENSE` zonder extensie (d.i. geen `LICENSE.txt` bijvoorbeeld)

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`, `__init__.py` en `LICENSE`. Maar het zou leuk zijn om een `README` te hebben en natuurlijk een pictogram om de plug-in weer te geven. Hieronder volgt een voorbeeld van hoe een `plugin.zip` eruit zou moeten zien.

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

HOOFDSTUK 17

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 algoritme 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 Modelontwerper of de Processing interface voor batchverwerking), en een snellere ontwikkelingstijd (omdat Processing een groot deel van het werk zal overnemen).

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

17.1 Maken vanaf niets

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

1. Installeer de plug-in **Plugin Builder**
2. Maak een nieuwe plug-in met de Plugin Builder. Wanneer de Plugin Builder u vraagt naar het te gebruiken sjabloon, selecteer dan “Processing provider”.
3. 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 algoritmes toe te voegen. Verwijs daarnaar voor meer meer informatie.

17.2 Een plug-in bijwerken

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

1. In uw bestand `metadata.txt` moet u een variabele toevoegen:

   ```
   hasProcessingProvider=yes
   ```

2. 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)
```

3. 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 algoritmes zal laten zien.

```
from qgis.core import QgsProcessingProvider
from qgis.PyQt.QtGui import QIcon

from .example_processing_algorithm import ExampleProcessingAlgorithm


class Provider(QgsProcessingProvider):

    """ The provider of our plugin. """

    def loadAlgorithms(self):
        """ Load each algorithm into the current provider. """
        self.addAlgorithm(ExampleProcessingAlgorithm())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())

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

   - `example_processing_algorithm.py` wat het voorbeeldbestand voor een algoritme bevat.

Kopieer/plak de inhoud van het bestand script template en werk dat naar behoefte bij.

U zou een soortgelijke boom moeten hebben als deze:

```
└── your_plugin_root_folder
    ├── __init__.py
    ├── LICENSE
    ├── metadata.txt
    └── processing_provider
        ├── example_processing_algorithm.py
        ├── __init__.py
        └── provider.py
```

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

HOOFDSTUK 18

Plug-in-lagen gebruiken

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

```
from qgis.core import (
    QgsPluginLayer,
    QgsPluginLayerType,
    QgsMapLayerRenderer,
    QgsApplication,
    QgsProject,
)

from qgis.PyQt.QtGui import QImage
```

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

18.1 Sub-klassen in QgsPluginLayer

Hieronder staat een voorbeeld van een minimale implementatie van QgsPluginLayer. Het is gebaseerd op de originele code van de voorbeeld plug-in Watermark.

De aangepaste renderer is het deel van de implementatie dat feitelijk tekent op het kaartvenster.

```
class WatermarkLayerRenderer(QgsMapLayerRenderer):

    def __init__(self, layerId, rendererContext):
        super().__init__(layerId, rendererContext)

    def render(self):
        image = QImage("/usr/share/icons/hicolor/128x128/apps/qgis.png")
        painter = self.renderContext().painter()
        painter.save()
        painter.drawImage(10, 10, image)
        painter.restore()
        return True

```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

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

    def createMapRenderer(self, rendererContext):
        return WatermarkLayerRenderer(self.id(), rendererContext)

    def setTransformContext(self, ct):
        pass

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

    def readXml(self, node, context):
        pass

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

De laag van de plug-in kan worden toegevoegd aan het project en aan het kaartvenster, net zoals elke andere laag:

```
plugin_layer = WatermarkPluginLayer()
QgsProject.instance().addMapLayer(plugin_layer)
```

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

```
class WatermarkPluginLayerType(QgsPluginLayerType):

    def __init__(self):
        super().__init__(WatermarkPluginLayer.LAYER_TYPE)

    def createLayer(self):
        return WatermarkPluginLayer()

    # You can also add GUI code for displaying custom information
    # in the layer properties
    def showLayerProperties(self, layer):
        pass


# Keep a reference to the instance in Python so it won't
# be garbage collected
plt =  WatermarkPluginLayerType()

assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)
```

HOOFDSTUK 19

Bibliotheek Netwerkanalyse

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

```
from qgis.core import (
  QgsVectorLayer,
  QgsPointXY,
)
```

De bibliotheek Netwerkanalyse kan worden gebruikt voor:

- rekenkundige grafieken maken uit geografische gegevens (polylijn vectorlagen)
- basismethoden implementeren 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.

19.1 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)

19.2 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: `QgsVectorLayerDirector`. 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 zijn alleen beschikbaar `QgsNetworkDistanceStrategy` strategie (die rekening houdt met de lengte van de route) en `QgsNetworkSpeedStrategy` (die ook rekening houdt met de snelheid). 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 analysis moeten importeren

```
from qgis.analysis 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 = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)

# 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 = QgsVectorLayerDirector(vectorLayer, 5, 'yes', '1', 'no',␣
↪QgsVectorLayerDirector.DirectionBoth)
```

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 = QgsVectorLayerDirector(vectorLayer,
                                  directionFieldId,
                                  directDirectionValue,
                                  reverseDirectionValue,
                                  bothDirectionValue,
                                  defaultDirection)
```

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

- `vectorLayer` — 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 een andere waarde heeft dan een van de hierboven gespecificeerde drie waarden. Mogelijke waarden zijn:
  - `QgsVectorLayerDirector.DirectionForward` — Éénrichting vooruit
  - `QgsVectorLayerDirector.DirectionBackward` — Éénrichting tegengesteld
  - `QgsVectorLayerDirector.DirectionBoth` — Twee richtingen

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

```
# The index of the field that contains information about the edge speed
attributeId = 1
# Default speed value
defaultValue = 50
# Conversion from speed to metric units ('1' means no conversion)
toMetricFactor = 1
strategy = QgsNetworkSpeedStrategy(attributeId, defaultValue, toMetricFactor)
```

En de director vertellen over deze strategie

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', 3)
director.addStrategy(strategy)
```

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 `True` (OTF gebruiken).
- `topologyTolerance` — topologische tolerantie. Standaardwaarde is 0.
- `ellipsoidID` — te gebruiken ellipsoïde. Standaard "WGS84".

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

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

```
startPoint = QgsPointXY(1179720.1871, 5419067.3507)
endPoint = QgsPointXY(1180616.0205, 5419745.7839)
```

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])
```

19.3 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 een 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 *n* de index van de inkomende rand of -1 als er geen inkomende randen zijn. In de tweede array bevat element *n* de afstand van de wortel van de boom tot het punt *n* of DOUBLE_MAX als het punt *n* 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.analysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
↪'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
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.edgeCount()):
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
  rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
  i = i + 1
```

Hetzelfde, maar met de methode `dijkstra()`

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')

director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
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(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
  rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())
```

19.3.1 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 T = B
while T != B
    add point T to path
    get incoming edge for point T
    look for point TT, that is start point of this edge
    assign T = TT
add point A 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 zou misschien een lijnenlaag willen laden en 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.analysis import *

from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
builder = QgsGraphBuilder(vectorLayer.sourceCrs())
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)

startPoint = QgsPointXY(1179661.925139,5419188.074362)
endPoint = QgsPointXY(1180942.970617,5420040.097560)

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

graph = builder.graph()
idxStart = graph.findVertex(tStart)

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

idxStart = tree.findVertex(tStart)
idxEnd = tree.findVertex(tStop)

if idxEnd == -1:
    raise Exception('No route!')

# Add last point
route = [tree.vertex(idxEnd).point()]

# Iterate the graph
while idxEnd != idxStart:
    edgeIds = tree.vertex(idxEnd).incomingEdges()
    if len(edgeIds) == 0:
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        break
    edge = tree.edge(edgeIds[0])
    route.insert(0, tree.vertex(edge.fromVertex()).point())
    idxEnd = edge.fromVertex()

# Display
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.green)

# This may require coordinate transformation if project's CRS
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

En hier is hetzelfde voorbeeld, maar voor de methode `dijkstra()`

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

from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)

builder = QgsGraphBuilder(vectorLayer.sourceCrs())

startPoint = QgsPointXY(1179661.925139,5419188.074362)
endPoint = QgsPointXY(1180942.970617,5420040.097560)

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

graph = builder.graph()
idxStart = graph.findVertex(tStart)
idxEnd = graph.findVertex(tStop)

(tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idxStart, 0)

if tree[idxEnd] == -1:
    raise Exception('No route!')

# Total cost
cost = costs[idxEnd]

# Add last point
route = [graph.vertex(idxEnd).point()]

# Iterate the graph
while idxEnd != idxStart:
    idxEnd = graph.edge(tree[idxEnd]).fromVertex()
    route.insert(0, graph.vertex(idxEnd).point())

# Display
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.red)
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
# This may require coordinate transformation if project's CRS
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

19.3.2 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

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())


pStart = QgsPointXY(1179661.925139, 5419188.074362)
delta = iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.green)
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPointXY(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 = 1500.0
i = 0
tree.reverse()

while i < len(cost):
    if cost[i] > r and tree[i] != -1:
        outVertexId = graph.edge(tree [i]).toVertex()
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

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

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

HOOFDSTUK 20

QGIS Server en Python

20.1 Introductie

Lees de QGIS-Server-manual om meer te leren over QGIS Server.

QGIS Server is drie verschillende dingen:

1. QGIS Server bibliotheek: een bibliotheek die een API verschaft voor het maken van webservices voor OGC
2. QGIS Server FCGI: een FCGI binaire toepassing `qgis_mapserv.fcgi` die, samen met een webserver, een set services van OGC implementeert (WMS, WFS, WCS etc.) en API's voor OGC (WFS3/OAPIF)
3. QGIS Development Server: een binaire toepassing voor een ontwikkelingsserver `qgis_mapserver` een set services van OGC implementeert (WMS, WFS, WCS etc.) en API's voor OGC (WFS3/OAPIF)

Dit hoofdstuk van het kookboek focust op het eerste onderwerp en door het gebruik van de API van QGIS Server uit te leggen, lat het zien hoe het mogelijk is Python te gebruiken om het gedrag van de server uit te breiden, te verbeteren of aan te passen of hoe de API van QGIS Server te gebruiken om QGIS server in een andere toepassing in te bedden.

Er zijn een aantal verschillende manieren waarmee u het gedrag van QGIS Server kunt wijzigen of de mogelijkheden ervan kan uitbreiden om nieuwe aangepaste services of API's aan te bieden. Dit zijn de belangrijkste scenario's die u zou kunnen tegenkomen:

- INBEDDEN → API van QGIS Server gebruiken vanuit een andere toepassing
- ZELFSTANDIG → QGIS Server uitvoeren als een zelfstandige service voor WSGI/HTTP
- FILTERS → QGIS Server verbeteren/aanpassen met plug-ins voor filters
- SERVICES → Een nieuwe *SERVICE* toevoegen
- OGC APIs → Een nieuwe *OGC API* toevoegen

Ingebedde en zelfstandige toepassingen vereisen het gebruiken van de Python API voor QGIS Server direct vanuit een ander script of toepassing voor Python. De resterende opties zijn beter geschikt voor wanneer u aangepaste mogelijkheden wilt toevoegen aan een standaard QGIS Server binaire toepassing (FCGI of ontwikkelingsserver): in dit geval dient u een plug-in voor Python voor de toepassing van de server te schrijven en uw aangepaste filters, services of API's te registreren.

20.2 Basisbeginselen Server API

De betrokken fundamentele klassen voor een typische toepassing van QGIS Server zijn:

- `QgsServer` de server instance (gewoonlijk één enkele instance voor de gehele levenscyclus van de toepassing)
- `QgsServerRequest` het object request object (gewoonlijk gemaakt voor elk verzoek)
- `QgsServer.handleRequest(request, response)` verwerkt het verzoek en vult het antwoord

De werkstroom voor een QGIS Server FCGI of ontwikkelingsserver kan als volgt worden samengevat:

```
initialize the QgsApplication
create the QgsServer
the main server loop waits forever for client requests:
    for each incoming request:
        create a QgsServerRequest request
        create a QgsServerResponse response
        call QgsServer.handleRequest(request, response)
            filter plugins may be executed
        send the output to the client
```

Binnen de methode `QgsServer.handleRequest(request, response)` worden de callbacks voor de filterplug-ins aangeroepen en `QgsServerRequest` en `QgsServerResponse` worden voor de plug-ins beschikbaar gemaakt via de klasse `QgsServerInterface`.

Waarschuwing: Klassen voor QGIS server zijn niet thread safe, u zou altijd een model voor multiverwerking of containers moeten gebruiken bij het bouwen van schaalbare toepassingen, gebaseerd op de API van QGIS Server.

20.3 Zelfstandig of inbedden

Voor zelfstandige toepassingen voor de server of inbedden dient u de hierboven vermelde klassen voor de server direct te gebruiken, verpak ze in een implementatie voor een webserver die alle interacties met de cliënt voor het protocol HTTP beheert.

Een minimaal voorbeeld voor het gebruiken van de API voor QGIS Server API (zonde het gedeelte voor HTTP) volgt:

```
from qgis.core import QgsApplication
from qgis.server import *
app = QgsApplication([], False)

# Create the server instance, it may be a single one that
# is reused on multiple requests
server = QgsServer()

# Create the request by specifying the full URL and an optional body
# (for example for POST requests)
request = QgsBufferServerRequest(
    'http://localhost:8081/?MAP=/qgis-server/projects/helloworld.qgs' +
    '&SERVICE=WMS&REQUEST=GetCapabilities')

# Create a response objects
response = QgsBufferServerResponse()

# Handle the request
server.handleRequest(request, response)

print(response.headers())
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
print(response.body().data().decode('utf8'))

app.exitQgis()
```

Hier staat een volledig voorbeeld van een zelfstandige toepassing, ontwikkeld voor het doorlopend testen van integraties in de opslagplaats voor de broncode van QGIS. Het laat een brede set van verschillende plug-ins voor filters en authenticatieschema's zien (niet bedoeld voor productie, omdat zij alleen voor testdoeleinden werden ontwikkeld, maar nog steeds interessant zijn om van te leren): qgis_wrapped_server.py

20.4 Server plug-ins

Server Python plug-ins worden geladen als de toepassing QGIS Server start en kunnen worden gebruikt om filters, services of API's te registreren.

De structuur van een server plug-in lijkt zeer veel op zijn collega voor de desktop, een object `QgsServerInterface` wordt beschikbaar gemaakt voor de plug-ins en de plug-ins kunnen één of meer aangepaste filters, services of API's registreren in het overeenkomende register door een van de methoden te gebruiken die worden blootgelegd door de interface van de server.

20.4.1 Server filter plug-ins

Filters zijn er in drie verschillende smaken en zij kunnen worden geïnstantieerd door een van de klassen hieronder te subklasseren en door de overeenkomende methode van `QgsServerInterface` aan te roepen:

Type filter	Basisklasse	registratie QgsServerInterface
I/O	`QgsServerFilter`	`registerFilter()`
Access Control	`QgsAccessControlFilter`	`registerAccessControl()`
Cache	`QgsServerCacheFilter`	`registerServerCache()`

I/O-filters

I/O-filters kunnen de in- en uitvoer van de server (het verzoek en het antwoord) van de bronservices (WMS, WFS etc.) aanpassen, wat het mogelijk maakt de werkstroom van de services te bewerken. Het is bijvoorbeeld mogelijk toegang tot de geselecteerde lagen te beperken, een XSL-stijlblad te injecteren in het antwoord van XML, om een watermerk toe te voegen aan een gemaakte afbeelding voor WMS, enzovoort.

Vanaf dit punt zou u het misschien nuttig kunnen vinden om eens snel te kijken naar de documentatie voor API server plug-ins.

Elk filter zou ten minste één van drie instructies moeten implementeren:

- `onRequestReady()`
- `onResponseComplete()`
- `onSendResponse()`

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).

Al deze methoden geven een Booleaanse waarde terug die aangeeft of de aanroep zou moeten worden doorgezet naar de opeenvolgende filters. Als één van deze methoden `False` teruggeeft stopt de keten, anders zal de aanroep worden doorgezet naar het volgende filter.

Hier is de pseudocode die weergeeft hoe de server een gewoon verzoek afhandelt en wanneer de callbacks van het filter worden aangeroepen:

```
for each incoming request:
    create GET/POST request handler
    pass request to an instance of QgsServerInterface
    call onRequestReady filters

    if there is not a response:
        if SERVICE is WMS/WFS/WCS:
            create WMS/WFS/WCS service
            call service's executeRequest
                possibly call onSendResponse for each chunk of bytes
                sent to the client by a streaming services (WFS)
        call onResponseComplete
    request handler sends the response to the client
```

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

onRequestReady

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).

onSendResponse

Dit wordt aangeroepen wanneer een gedeeltelijk antwoord wordt verwijderd uit de buffer voor het antwoord (d.i. naar **FCGI** `stdout` als de fcgi server wordt gebruikt) en vanaf daar naar de cliënt. Dit gebeurt als heel veel inhoud wordt gestroomd (zoals WFS GetFeature). In dat geval zou `onSendResponse()` meerdere keren kunnen worden aangeropen.

Onthoud dat als het antwoord niet wordt gestroomd, dan zal `onSendResponse()` helemaal niet worden aangeroepen.

In alle gevallen zal het laatste (of unieke) gedeelte worden verzonden aan de cliënt na het aanroepen van `onResponseComplete()`.

Teruggeven van `False` zal het doorsturen van gegevens naar de cliënt voorkomen. Dit is gewenst als een plug-in alle gedeelten uit een antwoord wil verzamelen en het antwoord onderzoeken of wijzigen in `onResponseComplete()`.

onResponseComplete

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 zal deze methode worden aangeroepen vóórdat het laatste (of unieke) gedeelte gegevens is verzonden aan de cliënt. Voor services voor streaming zouden meerdere aanroepen naar `onSendResponse()` gebeurd kunnen zijn.

`onResponseComplete()` 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 een watermerk aan een afbeelding van WMS toe te voegen).

Onthoud dat het teruggeven van `False` zal voorkomen dat de volgende plug-ins `onResponseComplete()` uitvoeren maar, in elk geval, voorkomen dat het antwoord wordt verzonden aan de cliënt.

Uitzonderingen opwerpen vanuit een plug-in

Er moet nog steeds wat werk worden gedaan aan dit onderwerp: de huidige implementatie kan onderscheid maken tussen afgehandelde en niet afgehandelde uitzonderingen door een eigenschap `QgsRequestHandler` in te stellen voor een instance van QgsMapServiceException. Op deze manier kan de belangrijkste code in C++ afgehandelde uitzonderingen voor Python opvangen 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.

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 alleen toegang tot een `QgsServerInterface` wanneer het wordt uitgevoerd binnen de context van de toepassing voor de QGIS Server.

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
```

Belangrijk: Alleen plug-ins die de metadata `server=True` hebben ingesteld zullen worden geladen en uitgevoerd door QGIS Server.

De voorbeeldplug-in qgis3-server-vagrant die hier wordt besproken (naast vele andere) is beschikbaar op GitHub, een klein aantal serverplug-ins zijn ook gepubliceerd in de officiële opslagplaats van plug-ins voor QGIS.

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*
```

__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` eruitziet:

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

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 instanties van een `QgsServerFilter`.

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

- `onRequestReady()`
- `onResponseComplete()`
- `onSendResponse()`

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

```
class HelloFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super().__init__(serverIface)

    def onRequestReady(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onRequestReady")
        return True

    def onSendResponse(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onSendResponse")
        return True

    def onResponseComplete(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onResponseComplete")
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
            request.clear()
            request.setResponseHeader('Content-type', 'text/plain')
            # Note that the content is of type "bytes"
            request.appendBody(b'HelloServer!')
        return True
```

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

```
class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(serverIface), 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:

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:

```
class ParamsFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(ParamsFilter, self).__init__(serverIface)

    def onRequestReady(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')
        return True

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.onResponseComplete")
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.onResponseComplete")
        return True
```

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/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms]␣
↪inserting pair SERVICE // HELLO into the parameter map
 src/mapserver/qgsserverfilter.cpp: 42: (onRequestReady) [0ms] QgsServerFilter␣
↪plugin default onRequestReady called
 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪SUCCESS - ParamsFilter.onResponseComplete
```

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).

Tip: Als u echt een aangepaste service wilt implementeren wordt aanbevolen om `QgsService` te subklasseren en uw service te registreren in `registerFilter()` door de `registerService(service)` daarvan aan te roepen

---

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:

```
from qgis.server import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

class WatermarkFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super().__init__(serverIface)

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (params.get('SERVICE').upper() == 'WMS' \
                and params.get('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.onResponseComplete: image␣
↪ready %s" % request.parameter("FORMAT"))
            # 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" if "png" in request.parameter("FORMAT") else
↪"JPG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
        return True
```

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 alleen PNG of JPG te ondersteunen.

Filters voor Access control

Filters voor Access control geven de ontwikkelaar fijnmazig beheer over tot welke lagen, objecten en attributen toegang kan worden verkregen, de volgende callbacks kunnen worden geïmplementeerd in een filter voor Access control:

- `layerFilterExpression(layer)`
- `layerFilterSubsetString(layer)`
- `layerPermissions(layer)`
- `authorizedLayerAttributes(layer, attributes)`
- `allowToEdit(layer, feature)`
- `cacheKey()`

Plug-inbestanden

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

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

__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:

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

AccessControl.py

```
class AccessControlFilter(QgsAccessControlFilter):

    def __init__(self, server_iface):
        super().__init__(server_iface)

    def layerFilterExpression(self, layer):
        """ Return an additional expression filter """
        return super().layerFilterExpression(layer)

    def layerFilterSubsetString(self, layer):
        """ Return an additional subset string (typically SQL) filter """
        return super().layerFilterSubsetString(layer)

    def layerPermissions(self, layer):
        """ Return the layer rights """
        return super().layerPermissions(layer)

```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    def authorizedLayerAttributes(self, layer, attributes):
        """ Return the authorised layer attributes """
        return super().authorizedLayerAttributes(layer, attributes)

    def allowToEdit(self, layer, feature):
        """ Are we authorised to modify the following geometry """
        return super().allowToEdit(layer, feature)

    def cacheKey(self):
        return super().cacheKey()

class AccessControlServer:

   def __init__(self, serverIface):
      """ Register AccessControlFilter """
      serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

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.

layerFilterExpression

Gebruikt om een expressie toe te voegen om de resultaten te beperken.

Bijvoorbeeld om te beperken tot de mogelijkheid waar het attribuut `role` gelijk is aan `user`.

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

layerFilterSubsetString

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

Bijvoorbeeld om te beperken tot de mogelijkheid waar het attribuut `role` gelijk is aan `user`.

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

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.

Bijvoorbeeld om alles te beperken tot toegang voor Alleen-lezen:

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

authorizedLayerAttributes

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

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

Bijvoorbeeld om het attribuut `role` te verbergen:

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

allowToEdit

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

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

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

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

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.

20.4.2 Aangepaste services

In QGIS Server zijn bronservices, zoals WMS, WFS en WCS, geïmplementeerd als subklassen van `QgsService`.

U kunt, om een nieuwe service te implementeren die zal worden uitgevoerd als de tekenreeks voor de query voor de parameter `SERVICE` overeenkomt met de naam voor de service, uw eigen `QgsService` implementeren en uw service registreren in `serviceRegistry()` door de methode `registerService(service)` daarvan aan te roepen

Hier is een voorbeeld van een aangepaste service, genaamd `CUSTOM`:

```
from qgis.server import QgsService
from qgis.core import QgsMessageLog

class CustomServiceService(QgsService):

    def __init__(self):
        QgsService.__init__(self)

    def name(self):
        return "CUSTOM"

```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    def version(self):
        return "1.0.0"

    def executeRequest(self, request, response, project):
        response.setStatusCode(200)
        QgsMessageLog.logMessage('Custom service executeRequest')
        response.write("Custom service executeRequest")


class CustomService():

    def __init__(self, serverIface):
        serverIface.serviceRegistry().registerService(CustomServiceService())
```

20.4.3 Aangepaste API's

In QGIS Server worden bron-API'S van OGC, zoals OAPIF, (alias WFS3) geïmplementeerd als collecties van subklassen van `QgsServerOgcApiHandler`, die zijn geregistreerd bij een instantie van `QgsServerOgcApi` (of zijn ouderklasse `QgsServerApi`).

U kunt, om een nieuwe API te implementeren die zal worden uitgevoerd als het pad voor de URL overeenkomt met een bepaalde URL, uw eigen instanties `QgsServerOgcApiHandler` implementeren, ze toevoegen aan een `QgsServerOgcApi` en de API registreren in het `serviceRegistry()` door de methode `registerApi(api)` daarvan aan te roepen.

Hier is een voorbeeld van een aangepaste API die zal worden uitgevoerd als de URL `/customapi` bevat:

```
import json
import os

from qgis.PyQt.QtCore import QBuffer, QIODevice, QTextStream, QRegularExpression
from qgis.server import (
    QgsServiceRegistry,
    QgsService,
    QgsServerFilter,
    QgsServerOgcApi,
    QgsServerQueryStringParameter,
    QgsServerOgcApiHandler,
)

from qgis.core import (
    QgsMessageLog,
    QgsJsonExporter,
    QgsCircle,
    QgsFeature,
    QgsPoint,
    QgsGeometry,
)


class CustomApiHandler(QgsServerOgcApiHandler):

    def __init__(self):
        super(CustomApiHandler, self).__init__()
        self.setContentTypes([QgsServerOgcApi.HTML, QgsServerOgcApi.JSON])

    def path(self):
        return QRegularExpression("/customapi")

    def operationId(self):
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        return "CustomApiXYCircle"

    def summary(self):
        return "Creates a circle around a point"

    def description(self):
        return "Creates a circle around a point"

    def linkTitle(self):
        return "Custom Api XY Circle"

    def linkType(self):
        return QgsServerOgcApi.data

    def handleRequest(self, context):
        """Simple Circle"""

        values = self.values(context)
        x = values['x']
        y = values['y']
        r = values['r']
        f = QgsFeature()
        f.setAttributes([x, y, r])
        f.setGeometry(QgsCircle(QgsPoint(x, y), r).toCircularString())
        exporter = QgsJsonExporter()
        self.write(json.loads(exporter.exportFeature(f)), context)

    def templatePath(self, context):
        # The template path is used to serve HTML content
        return os.path.join(os.path.dirname(__file__), 'circle.html')

    def parameters(self, context):
        return [QgsServerQueryStringParameter('x', True,␣
↪QgsServerQueryStringParameter.Type.Double, 'X coordinate'),
                QgsServerQueryStringParameter(
                    'y', True, QgsServerQueryStringParameter.Type.Double, 'Y␣
↪coordinate'),
                QgsServerQueryStringParameter('r', True,␣
↪QgsServerQueryStringParameter.Type.Double, 'radius')]


class CustomApi():

    def __init__(self, serverIface):
        api = QgsServerOgcApi(serverIface, '/customapi',
                              'custom api', 'a custom api', '1.1')
        handler = CustomApiHandler()
        api.registerHandler(handler)
        serverIface.serviceRegistry().registerApi(api)
```

HOOFDSTUK 21

Cheatsheet voor PyQGIS

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

```
from qgis.PyQt.QtCore import (
    QRectF,
)

from qgis.core import (
    Qgis,
    QgsProject,
    QgsLayerTreeModel,
)

from qgis.gui import (
    QgsLayerTreeView,
)
```

21.1 Gebruikersinterface

Uiterlijk wijzigingen

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
app.setStyleSheet(".QWidget {color: blue; background-color: yellow;}")
# You can even read the stylesheet from a file
with  open("testdata/file.qss") as qss_file_content:
    app.setStyleSheet(qss_file_content.read())
```

Pictogram en titel wijzigen

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon("/path/to/logo/file.png")
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("My QGIS")
```

21.2 Instellingen

Lijst QgsSettings ophalen

```
from qgis.core import QgsSettings

qs = QgsSettings()

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

21.3 Werkbalken

Werkbalk verwijderen

```
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])
```

21.4 Menu's

Menu verwijderen

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

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

21.5 Kaartvenster

Toegang tot kaartvenster

```
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.core import QgsSettings
# Set milliseconds (150 milliseconds)
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

21.6 Lagen

Vectorlaag toevoegen

```
layer = iface.addVectorLayer("testdata/data/data.gpkg|layername=airports",
↪"Airports layer", "ogr")
if not layer or not layer.isValid():
    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

```
1 from qgis.core import QgsVectorLayer
2 layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
3 QgsProject.instance().addMapLayer(layer)
4
5 layers_names = []
6 for layer in QgsProject.instance().mapLayers().values():
7     layers_names.append(layer.name())
8
9 print("layers TOC = {}".format(layers_names))
```

```
layers TOC = ['layer name you like']
```

Anders

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

```
layers TOC = ['layer name you like']
```

Laag zoeken op naam

```
from qgis.core import QgsProject

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

```
layer name you like
```

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 data reloading
layer.setAutoRefreshMode(Qgis.AutoRefreshMode.ReloadData)
```

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 QgsGeometry, QgsPointXY, QgsFeature

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

Objecten ophalen

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

```
<qgis._core.QgsFeature object at 0x7f45cc64b678>
```

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()))
```

```
10.000000, 10.000000
```

Geometrie verplaatsen

```
from qgis.core import QgsFeature, QgsGeometry
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POINT(7 45)")
geom.translate(1, 1)
poly.setGeometry(geom)
print(poly.geometry())
```

```
<QgsGeometry: Point (8 46)>
```

CRS instellen

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem('EPSG:4326'))
```

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
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
        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,-88.09 34.89,-88.39 30.34,-89.
↪57 30.18,-89.73 31,-91.63 30.99,-90.87 32.37,-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 vectorlagen uit GeoPackage laden

```
from qgis.core import QgsDataProvider

fileName = "testdata/sublayers.gpkg"
layer = QgsVectorLayer(fileName, "test", "ogr")
subLayers = layer.dataProvider().subLayers()

for subLayer in subLayers:
    name = subLayer.split(QgsDataProvider.SUBLAYER_SEPARATOR)[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 = 'https://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()
```

21.7 Inhoud

Toegang tot geselecteerde lagen

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

Contextmenu verwijderen

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

21.8 Uitgebreide inhoud

Bronknoop

```
from qgis.core import QgsVectorLayer, QgsProject, QgsLayerTreeLayer

root = QgsProject.instance().layerTreeRoot()
node_group = root.addGroup("My Group")

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

node_group.addLayer(layer)

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))
```

```
My Group
<class 'qgis._core.QgsLayerTreeGroup'>
False
True
```

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())
```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```

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())
```

```
- group: My Group
  - layer: layer name you like
```

Groep op naam zoeken

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

```
<QgsLayerTreeGroup: My Group>
```

Laag zoeken op ID

```
print(root.findLayer(layer.id()))
```

```
<QgsLayerTreeLayer: layer name you like>
```

Laag toevoegen

```
from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
# Remove it
QgsProject.instance().removeMapLayer(layer1)
```

Groep toevoegen

```
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
QgsProject.instance().mapLayersByName("layer name you like")[0]
```

Geladen laag verplaatsen

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

myLayer = root.findLayer(layer.id())
myClone = myLayer.clone()
parent = myLayer.parent()

myGroup = root.findGroup("My Group")
# Insert in first position
myGroup.insertChildNode(0, myClone)

parent.removeChildNode(myLayer)
```

Geladen laag naar een specifieke groep verplaatsen

```
QgsProject.instance().addMapLayer(layer, False)

root = QgsProject.instance().layerTreeRoot()
myGroup = root.findGroup("My Group")
myOriginalLayer = root.findLayer(layer.id())
myLayer = myOriginalLayer.clone()
myGroup.insertChildNode(0, myLayer)
parent.removeChildNode(myOriginalLayer)
```

Zichtbaarheid actieve laag schakelen

```
root = QgsProject.instance().layerTreeRoot()
node = root.findLayer(layer.id())
new_state = Qt.Checked if node.isVisible() == Qt.Unchecked else Qt.Unchecked
node.setItemVisibilityChecked(new_state)
```

Is groep geselecteerd

```
def isMyGroupSelected( groupName ):
    myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
    return myGroup in iface.layerTreeView().selectedNodes()

print(isMyGroupSelected( 'my group name' ))
```

```
False
```

Knoop uitbreiden

```
print(myGroup.isExpanded())
myGroup.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)
```

Laag verwijderen

```
root.removeLayer(layer)
```

Groep verwijderen

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

Nieuwe inhoudsopgave maken (TOC)

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

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")
```

21.9 Algoritmes voor Processing

Lijst algoritmes ophalen

```
from qgis.core import QgsApplication

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

```
QGIS (native c++):buffer --> Buffer
```

Help voor algoritmes ophalen

Willekeurige selectie

```
from qgis import processing
processing.algorithmHelp("native:buffer")
```

```
...
```

Het algoritme uitvoeren

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

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

```
Processing(0): Results: {'OUTPUT': 'output_d27a2008_970c_4687_b025_f057abbd7319'}
```

Hoeveel algoritmes zijn er?

```
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())
```

21.10 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)

```

(Vervolgt op volgende pagina)

(Vervolgd van vorige pagina)

```
    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()
```

21.11 Afdruklay-out

Afdruklay-out ophalen op naam

```
composerTitle = 'MyComposer' # Name of the composer

project = QgsProject.instance()
projectLayoutManager = project.layoutManager()
layout = projectLayoutManager.layoutByName(composerTitle)
```

21.12 Bronnen

- QGIS Python (PyQGIS) API
- QGIS C++ API
- StackOverFlow QGIS questions
- Script van Klas Karlsson