PyQGIS developer cookbook

Release 2.6

QGIS Project

22. May 2015

Inhoudsopgave

HOOFDSTUK 1

Introductie

Dit document is 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.

Beginnend met de uitgave 0.9, heeft QGIS optionele ondersteuning voor scripten met behulp van de taal Python. We hebben gekozen voor Python omdat het één van de meest favoriete talen voor scripten is. Bindingen voor PyQGIS zijn afhankelijk van SIP en PyQt4. De reden voor het gebruiken van SIP in plaats van het meer wijd gebruikte SWIG is dat de gehele code voor QGIS afhankelijk is van bibliotheken van Qt. Bindingen voor Python voor Qt (PyQt) worden ook gedaan met behulp van SIP en dat maakt een naadloze integratie van PyQGIS met PyQt mogelijk.

TODO: PyQGIS laten werken (handmatig compileren, probleemoplossing)

Er bestaan verscheidene manieren om bindingen van Python voor QGIS te gebruiken, zij worden in detail behandeld in de volgende gedeelten:

- opdrachten opgeven in de console voor Python in QGIS
- plug-ins in Python maken en gebruiken
- aangepaste toepassingen maken, gebaseerd op de API van QGIS

er is een verwijzing complete API voor QGIS dat de klassen uit de bibliotheken van QGIS documenteert. Pythonische API voor QGIS is nagenoeg identiek aan de API in C++.

Er bestaan enkele bronnen over het programmeren met PyQGIS op het QGIS blog. Bekijk QGIS tutorial ported to Python voor enkele voorbeelden van eenvoudige toepassingen van 3e partijen. Een goede bron voor het werken met plug-ins is om enkele plug-ins te downloaden vanaf plugin repository en hun code te bestuderen. Ook bevat de map `python/plugins/` in uw installatie van QGIS enkele plug-ins die u kunt gebruiken om te leren hoe een dergelijk plug-in wordt ontwikkeld en hoe enkele van de meest voorkomende taken uit te voeren

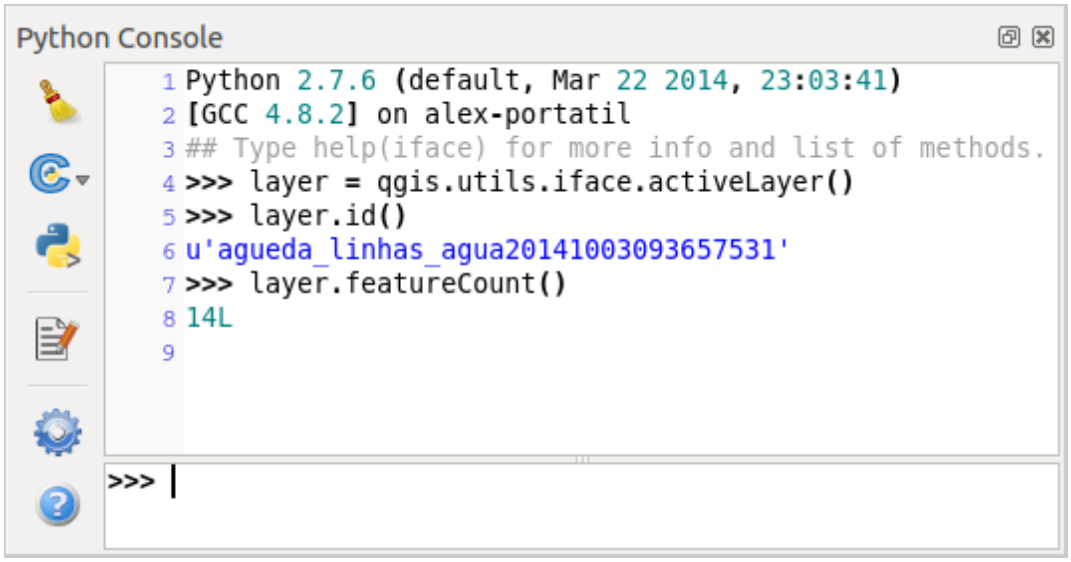
1.1 Python-console

Voor scripten is het mogelijk om voordeel te halen uit de geïntegreerde console voor Python. Deze kan geopend worden via het menu: *Plug-ins* → *Python Console*. De console opent als een non-modaal gebruiksvenster.

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 anatal objecten weer te geven. Voor interactie et de omgeving van QGIS is er de 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
```



Figuur 1.1: QGIS Python-console

Voor hen die de console vaak gebruiken, kan het handig zijn een sneltoets in te stellen voor het activeren van de console (in menu *Extra* → *Snelkoppelingen configureren...*)

1.2 Plug-ins in Python

QGIS maakt het mogelijk zijn functionaliteit te verbeteren met behulp van plug-ins. Dit was oorspronkelijk alleen mogelijk met de taal C++. Met de toevoeging van ondersteuning voor Python aan QGIS, is het ook mogelijk plug-ins te gebruiken die 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 Plugin Repositories voor verscheidene bronnen voor plug-ins.

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

1.3 Toepassingen in Python

Bij het verwerken van gegevens van GIS is het vaak handig om enkele scripts te maken voor het automatiseren van het proces in plaats van dezelfde taak steeds opnieuw te herhalen. 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 enige functionaliteit van GIS gebruikt — meten van enkele gegevens, exporteren van een kaart naar PDF of enige andere functionaliteit. De module `qgis.gui` geeft aanvullend verscheidene componenten voor een GUI, waarvan de meest belangrijke de widget voor het kaartvenster is die zeer eenvoudig kan worden opgenomen in de toepassing met ondersteuning voor zoomen, pannen en/of elke andere aangepaste gereedschappen voor de kaart.

1.3.1 PyQGIS gebruiken in aangepaste toepassing

opmerking: gebruik *niet* `qgis.py` als een naam voor uw testscript — Python zal niet in staat zijn de bindingen te importeren omdat de naam van het script die zal overschaduwen.

Als eerste dient u de module qgis te importeren, het pad voor QGIS in te stellen waar gezocht moet worden naar bronnen — database van projecties, providers etc. Wanneer u het prefix-pad instelt met als tweede argument ingesteld `True`, zal QGIS alle paden met standaard dir onder de map van de prefix initialiseren. Aanroepen van de functie `initQgis()` is belangrijk om QGIS te laten zoeken naar de beschikbare providers.

```
from qgis.core import *

# supply path to where is your qgis installed
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# load providers
QgsApplication.initQgis()
```

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 :-)

Wanneer u klaar bent met het gebruiken van de bibliotheek van QGIS, roep `exitQgis()` aan om er voor te zorgen dat alles wordt opgeruimd (bijv. het register van kaartlagen legen en lagen verwijderen):

```
QgsApplication.exitQgis()
```

1.3.2 Aangepaste toepassingen uitvoeren

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

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

Dit kan worden opgelost door de omgevingsvariabele `PYTHONPATH` in te stellen. In de volgende opdrachten zou `qgispath` moeten wordne 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**

Het pad naar de modules van PyQGIS is nu bekend, zij zijn echter afhankelijk van de bibliotheken `qgis_core` en `qgis_gui` (de modules van Python dienen slechts als verpakkingen). Het pad naar deze bibliotheken is meestal onbekend voor het besturingssysteem, dus krijgt u opnieuw een fout bij het importeren (het bericht kan variëren, afhankelijk van het systeem):

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

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

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

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

- eis dat de gebruiker QGIS op zijn platform installeert, voorafgaand aan het installeren van uw toepassing. Het installatieprogramma van de 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 - rol een zelfstandige toepassing uit op Windows en Mac OS X, laat voor Linux de installatie van QGIS over aan de gebruiker en diens pakketbeheer.

Lagen laden

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

2.1 Vectorlagen

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

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

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

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

- bibliotheek OGR (shapefiles en vele andere bestandsindelingen) — gegevensbron is het pad naar het bestand

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

- database PostGIS — gegevensbron is een string met alle informatie die nodig is om ene verbinding te maken naar de database van PostgreSQL. De klasse `QgsDataSourceURI` kan deze string voor u genereren. Onthoud dat QGIS moet worden gecompileerd met ondersteuning voor Postgres, anders is deze provider niet beschikbaar.

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

  vlayer = QgsVectorLayer(uri.uri(), "layer_name_you_like", "postgres")
  ```

- CSV of andere gescheiden tekstbestanden — om een bestand te openen met als scheidingsteken een puntkomma, met een veld “x” voor de X-coördinaat en veld “y” met de Y-coördinaat zou u iets zoals dit moeten gebruiken

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

  Opmerking: vanaf QGIS versie 1.7 is de string voor de provider 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 "X"- en "Y"-velden, en staat toe dat het coördinaten referentiesysteem wordt gespecificeerd. Bijvoorbeeld

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

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

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

- database SpatiaLite — ondersteund vanaf QGIS v1.1. Soortgelijk aan databases van PostGIS, `QgsDataSourceURI` kan wordne gebruikt voor het genereren van de identificatie van de gegevensbron

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

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

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

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

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

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

De uri kan worden gemaakt met behulp van de standaard bibliotheek `urllib`.

```
params = {
    'service': 'WFS',
    'version': '1.0.0',
    'request': 'GetFeature',
    'typename': 'union',
    'srsname': "EPSG:23030"
}
uri = 'http://localhost:8080/geoserver/wfs?' + urllib.unquote(urllib.urlencode(params))
```

2.2 Rasterlagen

Voor toegang tot rasterbestanden wordt de bibliotheek GDAL gebruikt. Het ondersteunt een breed scala aan bestandsindelingen. In het geval u problemen hebt met et 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 basisnaam om een raster uit een bestand te laden

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

Rasterlagen kunnen ook worden gemaakt vanuit een service voor WCS.

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

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

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

2.3 Register van kaartlagen

Als u de geopende lagen wilt gebruiken voor renderen, vergeet dan niet om ze toe te voegen aan het register van kaartlagen. Het register van kaartlagen wordt eigenaar van de lagen en er kan later toegang toe worden verkregen vanuit elk deel van de toepassing door middle van hun unieke ID. Als de laag wordt verwijderd uit het register van de kaartlagen, wordt hij ook verwijderd.

Een kaartlaag aan het register toevoegen:

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

Lagen worden bij het afsluiten automatisch vernietigd, als u echter de laag expliciet wilt verwijderen, gebruik dan:

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

TODO: Meer over het register van kaartlagen?

HOOFDSTUK 3

Rasterlagen gebruiken

Deze gedeelten vermelden verschillende bewerkingen die u kunt uitvoeren met rasterlagen.

3.1 Details laag

Een rasterlaag bestaat uit één of meer rasterbanden - er wordt naar verwezen als een ofwel enkelband of een multiband raster. Een band vertegenwoordigt een matrix van waarden. Gewoonlijk een kleurenafbeelding (bijv. luchtfoto) bestaat een raster uit een rode, blauwe en groene band. Lagen 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.

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

3.2 Tekenstijl

Wanneer een raster wordt geladen krijgt het een standaard stijl om te tekenen, gebaseerd op zijn type. Die kan worden gewijzigd, ofwel in de eigenschappen van de laag of programmatisch. De volgende stijlen voor tekenen bestaan:

Index	Constante: QgsRasterLater.X	Opmerking
1	SingleBandGray	Enkelbands afbeelding getekend als een bereik van grijswaarden
2	SingleBandPseudoColor	Enkelbands afbeelding getekend met behulp van een algoritme voor pseudokleur
3	PalettedColor	"Palet"-afbeelding getekend met behulp van ene kleurentabel
4	PalettedSingleBandGray	"Palet"-laag getekend in grijswaarden
5	PalettedSingleBandPseudoColor	"Palet"-laag tekenen met behulp van een algoritme voor pseudokleur
7	MultiBandSingleBandGray	Laag die 2 of meer banden bevat, maar een enkele band getekend als een bereik van grijswaarden
8	MultiBandSingleBandPseudoColor	Laag die 2 of meer banden bevat, maar een enkele band getekend met behulp van een algoritme voor pseudokleur
9	MultiBandColor	Laag die 2 of meer banden bevat, in kaart gebracht met de RGB-kleurruimte.

De huidige stijl van tekenen bevragen:

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

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

De volgende gedeelten leggen uit hoe de tekenstijl voor de laag te bevragen en aan te passen. U zou, na het aanbrengen van de wijzigingen, het bijwerken van het kaartvenster willen forceren, zie *Lagen vernieuwen*.

TODO: verbeteringen van contrast, transparantie (geen gegevens), gebruikergedefinieerde statistieken min/max, band

3.2.1 Enkelbands rasters

Zij worden standaard gerenderd in grijze kleuren. De tekenstijl wijzigen naar pseudokleur:

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

De `PseudoColorShader` is een basiskleurenbalk die lage waarden in blauw en hoge waarden in rood accentueert. Een andere, `FreakOutShader`, gebruikt meer kleurrijke kleuren en volgens de documentatie, zal het uw oma bang maken en uw honden laten huilen.

Er is ook `ColorRampShader` dat de kleuren in kaart brengt zoals gespecificeerd door zijn kleurenkaart. Het heeft drie modi voor de interpolatie van waarden:

- linear (`INTERPOLATED`): resulterende kleur wordt lineair geïnterpoleerd uit de items van de kleurenkaart boven en onder de actuele pixelwaarde
- discrete (`DISCRETE`): kleur wordt gebruikt uit het item voor de kleurenkaart met gelijke of hogere waarde
- exact (`EXACT`): kleur wordt niet geïnterpoleerd, alleen de pixels met een waarde gelijk aan die van de kleurenkaart worden getekend

Een interpolerende kleurenbalk instellen voor het bereik van groene naar gele kleur (voor pixelwaarden van 0 tot en met 255):

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

```
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

Gebruik, om terug te keren naar de standaard grijze niveaus:

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

3.2.2 Multiband rasters

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

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

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

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

3.3 Lagen vernieuwen

Indien u de symbologie van de laag wijzigt en er voor willen zorgen dat de wijzigingen onmiddellijk zichtbaar zijn voor de gebruiker, roep dan deze methoden aan:

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

De eerste aanroep zal er voor zorgen dat de afbeelding van de gerenderde laag in de cache wordt gewist in het geval dat caching van renderen is ingeschakeld. Deze functionaliteit is beschikbaar vanaf QGIS 1.4, in eerdere versies bestaat deze functie niet — om er voor te zorgen dat de code werkt met alle versies van QGIS, controleren we eerst of de methode bestaat.

De tweede aanroep verzendt het signaal dat een kaartvenster dat de laag bevat zal forceren het te vernieuwen.

Met rasterlagen van een WMS werken deze opdrachten niet. In dat geval dient u het expliciet te doen:

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

In het geval dat u symbologie voor lagen heeft gewijzigd (bekijk de gedeelten over raster- en vectorlagen om te zien hoe u dat doet), wilt u misschien QGIS forceren om de symbologie voor de lagen in de widget Lagenlijst (legenda) bij te werken. dat kan als volgt worden gedaan (`iface` is een instance van QgisInterface):

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

3.4 Waarden bevragen

Een query uitvoeren op waarden van banden van een rasterlaag op een gespecificeerd punt:

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

De methode `results` geeft in dit geval een woordenboek terug, met indices van banden als sleutels, en bandwaarden als waarden.

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

Vectorlagen gebruiken

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

4.1 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

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

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

    # fetch attributes
    attrs = feature.attributes()

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

Naar attributen kan worden verwezen door middel van een index.

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

4.1.1 Itereren over geselecteerde objecten

Methoden voor het gemak.

Voor bovenstaande gevallen en voor het geval dat u selecteren in een vectorlaag overweegt in het geval die bestaat, kunt u de methode `features()` gebruiken uit de ingebouwde plug-in Processing, en wel als volgt:

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

Hiermee worden alle objecten in een laag doorlopen indien er geen selectie actief is, of anders over de geselecteerde objecten.

als u alleen objecten moet selecteren, kunt u de methode `selectedFeatures()` gebruiken uit de vectorlaag:

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

4.1.2 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 gegeven gebied, dan dient een object `QgsFeatureRequest` te worden toegevoegd aan de aanroep `getFeatures()`. Hier is een voorbeeld:

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

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

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

4.2 Vectorlagen bewerken

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

```
caps = layer.dataProvider().capabilities()
```

Bij het gebruiken van de volgende methodes voor het bewerken van vectorlagen worden de wijzigingen direct opgeslagen in de onderliggende gegevensbron (bestanden, 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*.

4.2.1 Objecten toevoegen

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

```
if caps & QgsVectorDataProvider.AddFeatures:
  feat = QgsFeature()
  feat.addAttribute(0, 'hello')
  feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
  (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

4.2.2 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])
```

4.2.3 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.fromPoint(QgsPoint(111,222))
  layer.dataProvider().changeGeometryValues({ fid : geom })
```

4.2.4 Velden toevoegen en verwijderen

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

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

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

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

```
layer.updateFields()
```

4.3 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.

Gebruik `isEditing()` om te zien of een laag in de modus Bewerken staat — de functies voor bewerken werken alleen als de modus Bewerken is ingeschakeld. Gebruiken van functies voor bewerken

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

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

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

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*.) Hoe de functionaliteit Ongedaan maken te gebruiken

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

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

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

`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.

Er is de methode `startEditing()` om de modus Bewerken te starten, om te stoppen met bewerken zijn er `commitChanges()` en `rollback()` — echter, normaal gesproken zou u deze methoden niet nodig hebben en laat deze functionaliteit te worden geactiveerd door de gebruiker.

4.4 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.

1. ruimtelijke index maken — de volgende code maakt een lege index

   ```
   index = QgsSpatialIndex()
   ```

2. objecten aan index toevoegen — 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 `nextFeature()` van de provider

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

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

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

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

4.5 Vectorlagen schrijven

U kunt bestanden voor vectorlagen schrijven met behulp van de klasse `QgsVectorFileWriter`. Het ondersteunt elke andere soort vectorbestand dat wordt ondersteund door OGR (shapefiles, GeoJSON, KML en andere).

Er zijn twee mogelijkheden voor het exporteren van een vectorlaag:

- vanuit een instance van `QgsVectorLayer`

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

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

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

  De derde parameter specificeert de codering voor uit te voeren tekst. Alleen enkele stuurprogramma's hebben dit nodig voor een juiste verwerking - shapefiles zijn er daar één van — in het geval u echter geen internationale tekens gebruikt, hoeft u zich niet echt druk te maken over de codering. De vierde parameter die we hebben gelaten als `None` kan het doel-CRS specificeren — als een geldige instance van `QgsCoordinateReferenceSystem` wordt doorgegeven, wordt de laag naar dat CRS getransformeerd.

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

- direct uit objecten

  ```
  # define fields for feature attributes. A list of QgsField objects is needed
  fields = [QgsField("first", QVariant.Int),
            QgsField("second", QVariant.String)]

  # create an instance of vector file writer, which will create the vector file.
  # Arguments:
  ```

```
# 1. path to new file (will fail if exists already)
# 2. encoding of the attributes
# 3. field map
# 4. geometry type - from WKBTYPE enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Sh

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk (optional)
del writer
```

4.6 Memory-provider

Memory-provider is bedoeld om te worden gebruikt door voornamelijk plug-in of ontwikkelaars voor 3e partijen. Het slaat geen gegevens op op de schijf, 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"`, of `"MultiPolygon"`.

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

crs=definition 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

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

# add fields
```

```
pr.addAttributes([QgsField("name", QVariant.String),
                  QgsField("age",  QVariant.Int),
                  QgsField("size", QVariant.Double)])

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

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

Laten we tenslotte controleren of alles goed ging

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

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

4.7 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.rendererV2()
```

En met die verwijzing, laten we het een beetje verkennen

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

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

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

Er kunnen ook enkele aangepaste typen renderer zijn, dus doe nooit de aanname dat slechts deze typen beschikbaar zijn. U kunt singleton `QgsRendererV2Registry` bevragen om de huidige beschikbare renderers te achterhalen.

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

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

4.7.1 Renderer Enkel symbool

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

4.7.2 Renderer symbool Categoriën

U kunt de naam van het attribuut, dat gebruikt wordt voor de classificatie, bevragen en instellen: gebruik de methoden: `classAttribute()` en `setClassAttribute()`.

Een lijst categorieën verkrijgen

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

Waar `value()` de waarde is die wordt gebruikt voor de verdeling in 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()`.

4.7.3 Renderer symbool Gradueel

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

Meer te weten komen over gebruikte bereiken in de renderer

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

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

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

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
```

```
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

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

4.7.4 Werken met symbolen

Voor het weergeven van symbolen is er de basisklasse `QgsSymbolV2` met drie afgeleide klassen:

- `QgsMarkerSymbolV2` — voor objecten punt
- `QgsLineSymbolV2` — voor objecten lijn
- `QgsFillSymbolV2` — voor objecten polygoon

Elk symbool bestaat uit één of meer symboollagen (klassen afgeleid van `QgsSymbolLayerV2`). De symboollagen doen de actuele rendering, de symboolklasse zelf dient alleen als een container voor de symboollagen.

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

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

Gebruik de methode `color()` om de kleur van het symbool vast te stellen en `setColor()` om die kleur te wijzigen. Met aanvullende markeringssymbolen kunt u vragen naar de grootte en rotatie van het symbool met de methoden `size()` en `angle()`, voor lijnsymbolen is er de methode `width()` die de dikte van de lijn teruggeeft.

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

Werken met symboollagen

Zoals eerder gezegd bepalen symboollagen (subklassen van `QgsSymbolLayerV2`) 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 kunt maken voor een bepaalde klasse van een symboollaag, verkrijgen op deze manier

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

Uitvoer

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

Klasse `QgsSymbolLayerV2Registry` beheert een database van alle beschikbare typen symboollagen.

Gebruik zijn methode `properties()` om toegang te verkrijgen tot de gegevens van de symbollaag, 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()`, `width()` met hun tegenhangers om ze in te stellen. Natuurlijk zijn grootte en hoek alleen beschikbaar voor symboollagen voor markeringen en breedte voor lijn-symboollagen.

Aangepaste typen voor symboollagen maken

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

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

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

  def layerType(self):
    return "FooMarker"

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

  def startRender(self, context):
    pass

  def stopRender(self, context):
    pass

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

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

De methode `layerType()` bepaalt de naam van de symboollaag, die moet uniek zijn voor alle symboollagen. Eigenschappen worden 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()` die het renderen uitvoert. 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, resp. `renderPolygon()` welke een lijst van punten op de buitenste ring als een 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

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

    self.layer = None

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

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

  def symbolLayer(self):
    return self.layer

  def radiusChanged(self, value):
    self.layer.radius = value
    self.emit(SIGNAL("changed()"))
```

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

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

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

We zullen metadata moeten maken voor de symboollaag

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

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

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

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

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

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. `createSymbolLayer()` zorgt voor het maken van een instance van de symboollaag met attributen die zijn gespecificeerd in het woordenboek *props*. (Let op: de sleutels zijn QString instances, geen "str"-objecten). En er is de methode `createSymbolLayerWidget()` die 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.

4.7.5 Aangepaste renderers maken

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

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

```
import random

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

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

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

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

  def usedAttributes(self):
    return []

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

De constructor van de bovenliggende klasse `QgsFeatureRendererV2` heeft de naam van de renderer nodig (moet uniek 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 ene lijst met veldnamen teruggeven waarvan de renderer verwacht dat die aanwezig is. Tenslotte zou de functie `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 `QgsRendererV2Widget`. De volgende voorbeeldcode maakt een knop die de gebruiker in staat stelt het symbool in te stellen van het eerste symbool

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

```
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2("Color 1")
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

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

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

De constructor ontvangt instances van de actieve laag (`QgsVectorLayer`), de globale opmaak (`QgsStyleV2`) en huidige renderer. Indien er geen renderer is of de renderer heeft een andere type, zal die worden vervangen door onze nieuwe renderer, anders zullen we de huidige renderer gebruiken (die al het type heeft dat we nodig hebben). De inhoud van de widget zou moeten worden bijgewerkt om de huidige staat van de renderer weer te geven. Wanneer het dialoogvenster van de renderer wordt geaccepteerd, wordt de methode voor de widget `renderer()` aangeroepen om de huidige renderer te verkrijgen — het 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

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

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

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

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

U kunt, om een pictogram te associëren met de renderer, die toewijzen in de constructor `QgsRendererV2AbstractMetadata` als een derde (optioneel) argument — de basis klassse-constructor in de functie `__init__()` van de RandomRendererMetadata wordt

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

Het pictogram kan ook op een later tijdstip worden geassocieerd met behulp van 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 (PyQt4 bevat .qrc compiler voor Python).

4.8 Meer onderwerpen

TODO: maken/aanpassen van symbolen die werken met stijl (`QgsStyleV2`) werken met kleurbalken (`QgsVectorColorRampV2`) op regels gebaseerde-renderer (zie deze blogpost) die de symboollaag en registreren van de renderer verkent

HOOFDSTUK 5

Afhandeling van geometrie

Naar punten, lijnen en polygonen die een ruimtelijk object weergeven wordt gewoonlijk verwezen als geometrieën. In QGIS worden zij weergegeven door de klasse `QgsGeometry`. Alle mogelijke typen geometrie worden netjes weergegeven op de pagina JTS discussion.

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.

5.1 Construeren van geometrie

Er zijn verscheidene opties voor het maken van een geometrie:

- uit coördinaten

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

  Coördinaten worden opgegeven met behulp van de klasse `QgsPoint`.

  Polylijnen (lijnen) worden weergegeven als een lijst met punten. 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.

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

- uit bekende tekst (WKT)

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

- uit bekende binaire (WKB)

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

5.2 Toegang tot geometrie

Als eerste zou u het type geometrie moeten zoeken, de methode `wkbType()` is degene om te gebruiken — het geeft een waarde uit de enumeratie QGis.WkbType terug

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

Als alternatief kan men de methode `type()` gebruiken die een waarde teruggeeft uit de enumeratie QGis.GeometryType. Er is ook een hulpfunctie `isMultipart()` om na te gaan of een geometrie uit meerdere delen bestaat of niet.

Voor elk type vector zijn er functies voor toegang om informatie uit de geometrie op te halen. Hoe deze te gebruiken

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

Opmerking: 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()`, `asMultiPolygon()`.

5.3 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 (`union()`, `difference()`, ...). Het kan ook geometrische eigenschappen van geometrieën berekenen, zoals gebied (in het geval van polygonen) of lengten (voor polygonen en lijnen)

Hier is een klein voorbeeld dat het doorlopen van de objecten op een laag combineert met het uitvoeren van enkele geometrische berekeningen, gebaseerd op hun geometrieën.

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

Gebieden en perimeters gouden geen rekening met het CRS indien wordt berekend met behulp van deze methoden uit de klasse `QgsGeometry`. Voor een meer krachtige berekening van gebied en afstand, kan de klasse `QgsDistanceArea` worden gebruikt. Als projecties zijn uitgeschakeld, zullen berekeningen vlak zijn, anders zullen zij op de ellipsoïde worden uitgevoerd. Wanneer niet expliciet een ellipsoïde is ingesteld, worden parameters voor WGS84 gebruikt voor berekeningen.

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

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

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

Aanvullende informatie kan worden gevonden in de volgende bronnen:

- Transformeren van geometrie: Algoritme Opnieuw projecteren
- Afstand en gebied met behulp van de klasse `QgsDistanceArea`: Algoritme Afstandsmatrix
- Algoritme Van meerdelig naar ééndelig

HOOFDSTUK 6

Ondersteuning van projecties

6.1 Coördinaten ReferentieSystemen

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

- Specificeer een CRS aan de hand van een code (ID)

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

  QGIS gebruikt drie verschillende ID's voor elk referentiesysteem:

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

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

- specificeer een CRS middels 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)
  ```

- maak een ongeldige CRS en gebruik deze in de `create*()` in het volgende voorbeeld wordt een Proj4 regel gebruikt om de projectie te initialiseren.

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

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

Voor het initialiseren van ruimtelijke referentiesystemen moet QGIS, de instellingen opzoeken in een database `srs.db`. Bij het maken van een eigen applicatie dienen de paden te worden ingesteld met `QgsApplication.setPrefixPath()` anders zal het zoeken naar de database mislukken. Als opdrachten worden uitgevoerd vanuit de Python console in QGIS of vanuit een plug-in dan staat dit pad al goed ingesteld.

Informatie van ruimtelijke referentiesystemen benaderen

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

```
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

6.2 Projecties

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

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

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

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

HOOFDSTUK 7

Kaartvenster gebruiken

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.

Kaartvenster wordt geïmplementeerd als 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.

Wanneer de kaart wordt verschoven, in of uit wordt gezoomd (of een andere actie die een verversing activeert), wordt de kaart opnieuw gerenderd binnen de huidige inhoud. De lagen worden tot een afbeelding gerenderd (met behulp van de klasse `QgsMapRenderer`) en ie afbeelding wordt dan weergegeven in het kaartvenster. Het item voor de afbeelding (in termen van de grafische weergave van het framework van Qt) dat verantwoordelijk is voor het weergeven van de kaart is de klasse `QgsMapCanvasMap`. Deze klasse beheert ook het vernieuwen van de gerenderde kaart. Naast het item dat 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 kasisobjecten `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

7.1 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 bij het gebruiken van .ui-bestanden en Qt Designer.

De functionaliteit `pyuic4` 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 `PyQt4.QtCore` en `Qt.white` is één van de voorgedefinieerde instances `QColor`.)

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

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

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

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

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

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

7.2 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 van 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 PyQt4.QtGui import QAction, QMainWindow
from PyQt4.QtCore import SIGNAL, Qt, QString

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

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

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

    self.setCentralWidget(self.canvas)

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

```
    actionPan = QAction(QString("Pan"), self)

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

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

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

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

    self.pan()

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

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

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

U kunt de bovenstaande code plaatsen in een bestand, bijv. `mywnd.py` en het uitproberen in de console van Python binnen QGIS. Deze code zal de huidige geselecteerde laag in een nieuw gemaakt kaartvenster plaatsen

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

Zorg er echter voor dat het bestand `mywnd.py` is geplaatst binnen het zoekpad van Python (`sys.path`). Als dat niet zo is kunt u het eenvoudig toevoegen: `sys.path.insert(0, '/my/path')` — anders zal het argument voor het importeren falen, het vindt de module niet.

7.3 Elastieken banden en markeringen voor punten

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

Een polylijn weergeven

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

Een polygoon weregeven

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

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

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

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

De items voor het kaartvenster zij 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, de klasse `QgsVertexMarker` is echter beter geschikt hiervoor (`QgsRubberBand` zou alleen een rechthoek rondom het gewenste punt tekenen). Hoe de markering voor punten te gebruiken

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

Dit zal een rood kruis tekenen op de positie [0,0]. 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)
```

Voor het tijdelijk verbergen van markeringen voor punten en ze uit het kaartvenster te verwijderen, is hetzelfde van toepassing als voor elastieken banden.

7.4 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 ervan 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, QGis.Polygon)
      self.rubberBand.setColor(Qt.red)
      self.rubberBand.setWidth(1)
      self.reset()
```

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

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

def canvasReleaseEvent(self, e):
    self.isEmittingPoint = False
    r = self.rectangle()
    if r is not None:
      print "Rectangle:", r.xMin(), r.yMin(), r.xMax(), r.yMax()

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

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

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

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

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

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

    return QgsRectangle(self.startPoint, self.endPoint)

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

7.5 Aangepaste items voor het kaartvenster schrijven

TODO: hoe een item voor het kaartvenster te maken

HOOFDSTUK 8

Kaart renderen en afdrukken

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 `QgsMapRenderer` of produceer een meer fijn afgestemde uitvoer door de kaart samen te stellen met behulp van de klasse `QgsComposition` en diens vrienden.

8.1 Eenvoudig renderen

Render enkele lagen met behulp van `QgsMapRenderer` — maak een doel-kleurapparaat (`QImage`, `QPainter` etc.), stel de set van kaartlagen in, bereik, grootte van de uitvoer en voer het renderen uit

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

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

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

render = QgsMapRenderer()

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

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

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

# do the rendering
render.render(p)

p.end()

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

8.2 Uitvoer met behulp van Printvormgeving

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

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

De centrale klasse van de printvormgeving is `QgsComposition` die is afgeleid van `QGraphicsScene`. Laten we er een maken

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

Onthoud dat de Printvormgeving een instance heeft van `QgsMapRenderer`. In de code gaan we er van uit dat het binnen de toepassing QGIS wordt uitgevoerd en dus de renderer voor de kaart gebruiken van het kaartvenster. De compositie gebruikt verscheidene parameters van de renderer van de kaart, meest belangrijke daarvan zijn de standaard set kaartlagen en het huidige bereik. Bij het gebruiken van printvormgeving in een zelfstandige toepassing kunt u uw eigen instance van de renderer voor de kaart maken op dezelfde manier zoals die welke in het gedeelte hierboven wordt weergegeven en die doorgeven aan de compositie.

Het is mogelijk om verscheidene elementen (map, label, ...) toe te voegen aan de compositie — deze elementen moeten afstammen van de klasse `QgsComposerItem`. De huidige ondersteunde items zijn:

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

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

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

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

- legenda

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

- schaalbalk

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

- pijl
- afbeelding
- vorm

- tabel

Standaard hebben de nieuw gemaakte items voor Printvormgeving de positie nul (linkerbovenhoek van de pagina) en de grootte nul. De positie en grootte worden altijd in millimeters gemeten

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

Een kader wordt standaard rondom elk item getekend. Hoe dat kader te verwijderen

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

Naast het handmatig maken van items voor Printvormgeving, heeft QGIS ondersteuning voor sjablonen van Printvormgeving wat in essentie composities zijn met al hun items, opgeslagen als een bestand .qpt (met syntaxis XML). Helaas is deze functionaliteit nog niet beschikbaar in de API.

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

De standaard instellingen voor uitvoer van de compositie zijn paginagrootte A4 en resolutie 300 DPI. U kunt ze wijzigen, indien nodig. De grootte van het papier wordt gespecificeerd in millimeters

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

8.2.1 Naar een rasterafbeelding uitvoeren

Het volgende fragment code geeft weer hoe een compositie te renderen naar een rasterafbeelding

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

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

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

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

8.2.2 Naar PDF uitvoeren

Het volgende fragment code rendert een compositie naar een PDF-bestand

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

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

Expressies, filteren en waarden berekenen

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).

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`
- geometrische functies: `$area`, `$length`
- functies voor conversie: `to int`, `to real`, `to string`

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`

9.1 Parsen van expressies

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

9.2 Evalueren van expressies

9.2.1 Basisexpressies

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

9.2.2 Expressies met objecten

Het voorbeeld evalueert de opgegeven expressie ten opzichte van een object. "Kolom" is de naam van het veld in de laag.

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

U kunt ook `QgsExpression.prepare()` gebruiken als u mmer dan één object wilt controleren. Het gebruiken van `QgsExpression.prepare()` zal de snelheid verhogen die de evaluatie nodig heeft om te worden uitgevoerd.

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

9.2.3 Fouten afhandelen

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

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

print value
```

9.3 Voorbeelden

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

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

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

Instellingen lezen en opslaan

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

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

We kunnen onderscheid maken tussen de verscheidene typen instellingen:

- **globale instellingen** — zij zijn gebonden aan de gebruiker op een bepaalde machine. QGIS slaat zelf heel veel globale instellingen op, bijvoorbeeld, de grootte van het hoofdvenster of de standaard tolerantie voor snappen. Deze functionaliteit wordt direct verschaft door het framework Qt door middel van de klasse QSettings. Standaard slaat deze klasse instellingen op in de "eigen" manier van het systeem voor het opslaan van instellingen, dat is — registrer (op Windows), bestand .plist (op Mac OS X) of bestand .ini (op Unix). De QSettings documentation is zeer uitgebreid, dus zullen we slechts een eenvoudig voorbeeld geven

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

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

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

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

  ```
  proj = QgsProject.instance()

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

  # read values
  ```

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

Zoals u kunt zien wordt de methode `writeEntry()` gebruikt voor alle gegevenstypen, mer 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 een projectbestand, dus als de gebruiker het project opnieuw opent, zijn de aan de laag gerelateerde instellingen weer aanwezig. Deze functionaliteit is toegevoegd in QGIS v1.4. De API is soortgelijk aan QSettings — het ontvangt en geeft instances van QVariant terug

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

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

Communiceren met de gebruiker

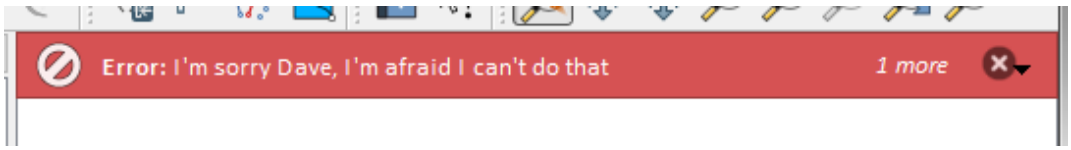
Dit gedeelte geeft enkele methoden en elementen weer die zouden moeten worden gebruikt om te communiceren met de gebruiker, om consistentie in de gebruikersinterface te behouden.

11.1 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

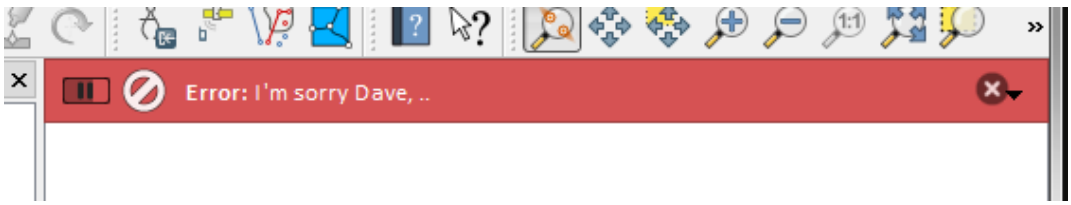
```
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that", level=QgsMes
```



Figuur 11.1: QGIS berichtenbalk

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

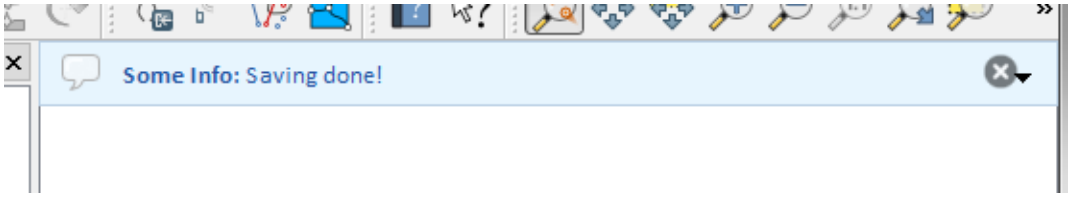
```
iface.messageBar().pushMessage("Error", ""Ooops, the plugin is not working as it should", level=Q
```



Figuur 11.2: 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 constanten `QgsMessageBar.WARNING` en `QgsMessageBar.INFO`.

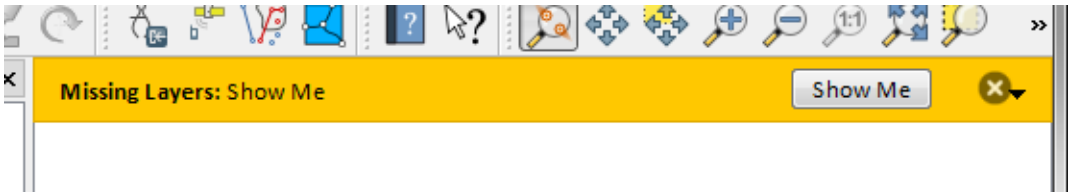
Widgets kunnen aan de berichtenbalk worden toegevoegd, zoals bijvoorbeeld een knop om meer informatie weer te geven



Figuur 11.3: QGIS berichtenbalk (info)

```
def showError():
    pass

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



Figuur 11.4: QGIS berichtenbalk met een knop

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

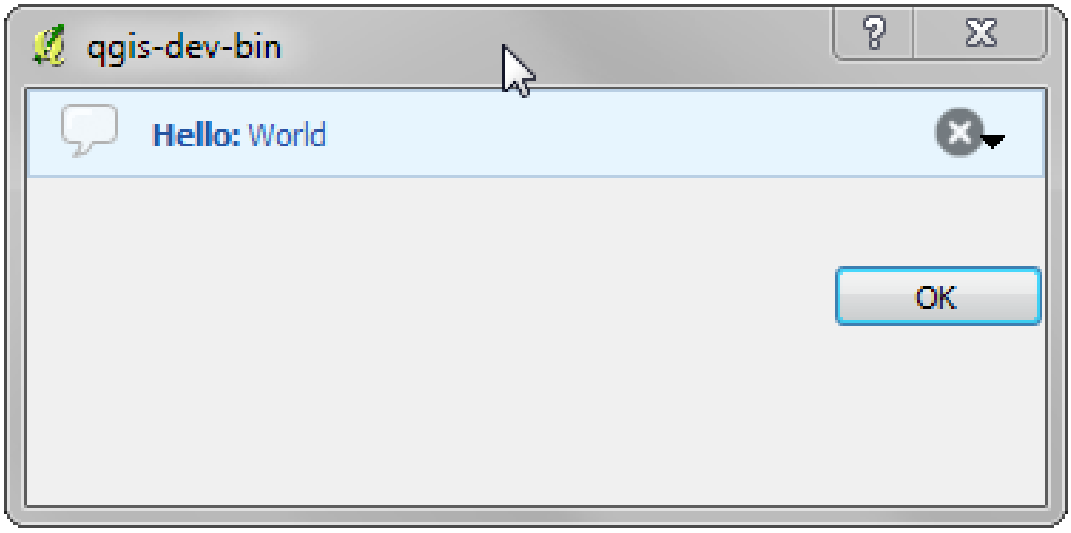
```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0,0,0,0)
        self.buttonbox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

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

11.2 Voortgang weergeven

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

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
```



Figuur 11.5: QGIS berichtenbalk in aangepast dialoogvenster

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

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

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

11.3 Loggen

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

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

HOOFDSTUK 12

Python plug-ins ontwikkelen

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

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

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

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

Stappen:

1. *Idee*: Weet wat u met uw nieuwe plug-in voor QGIS wilt gaan doen. Waarom doet u dat? Welk probleem wilt u oplossen? Is er al een andere plug-in voor dat probleem?
2. *Bestanden maken*: Maak de hieronder beschreven bestanden. Een beginpunt (`__init__.py`). Vul *Metadata van de plug-in* (`metadata.txt`) in. Een hoofdgedeelte voor een plug-in in Python (`mainplugin.py`). Een formulier in QT-Designer (`form.ui`), met zijn `resources.qrc`.
3. *Code schrijven*: Schrijf de code in `mainplugin.py`
4. *Testen*: Sluit en heropen QGIS en importeer uw plug-in opnieuw. Controleer of alles OK is.
5. *Publiceren*: Publiceer uw plug-in in de opslagplaats van QGIS of maak uw eigen opslagplaats als een "arsenaal" van persoonlijke "wapens voor GIS".

12.1 Een plug-in schrijven

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

12.1.1 Plug-inbestanden

Hier is de mappenstructuur van uw voorbeeld-plug-in

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

Wat is de betekenis van de bestanden:

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

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

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

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

12.2 Inhoud van de plug-in

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

12.2.1 Metadata van de plug-in

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

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

Naam van de metadata	Vereist	Opmerkingen
name	Ja	een korte string die de naam van de plug-in bevat
qgisMinimumVersion	Ja	gestippelde notatie van de minimale versie van QGIS
qgisMaximumVersion	Nee	gestippelde notatie van de maximale versie van QGIS
description	Ja	korte tekst die de plug-in beschrijft, geen HTML toegestaan
about	Nee	langere tekst die de plug-in tot in detail beschrijft, geen HTML toegestaan
version	Ja	korte string met de versie in gestippelde notatie
author	Ja	author name
email	Ja	e-mail van de auteur, zal *niet* worden weergegeven op de website
changelog	Nee	string, mag meerdere regels zijn, geen HTML toegestaan
experimental	Nee	Booleaanse vlag, *True* of *False*
deprecated	Nee	Booleaanse vlag, *True* of *False*, is van toepassing op de gehele plug-in en niet alleen op de geüploade versie
tags	Nee	kommagescheiden lijst, spaties zijn binnen de individuele tags toegestaan
homepage	Nee	een geldige URL die verwijst naar de startpagina voor uw plug-in
repository	Nee	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)
category	Nee	één van *Raster*, *Vector*, *Database* of *Web*

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

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

```
category=Raster
```

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

Een voorbeeld voor dit metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
; end of mandatory metadata

; start of optional metadata
category=Raster
changelog=The changelog lists the plugin versions
```

```
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

; Tags are in comma separated value format, spaces are allowed within the
; tag name.
; Tags should be in English language. Please also check for existing tags and
; synonyms before creating a new one.
tags=wkt,raster,hello world

; these metadata can be empty, they will eventually become mandatory.
homepage=http://www.itopen.it
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
icon=icon.png

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

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

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

12.2.2 __init__.py

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

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

## any other initialisation needed
```

12.2.3 mainPlugin.py

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

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

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

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

```
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

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

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

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

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

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

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

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

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

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

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

Alle hebben dezelfde syntaxis als de methode `addPluginToMenu()`.

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

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

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

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

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

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

12.2.4 Bronbestand

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

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

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

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

En dat is alles... niets gecompliceerds :)

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

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

12.3 Documentatie

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

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

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

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.

13.1 Een opmerking voor het configureren van uw IDE op Windows

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

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

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

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

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

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

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

Volg deze stappen om het batch-bestand te maken.

- Zoek naar de map waar het bestand file:*qgis_core.dll* is geplaatst. Normaal gesproken is dit `C:OSGeo4Wappsqgisbin`, maar als u uw eigen toepassing in QGIS compileerde is het in de map waar uw het bouwde in `output/bin/RelWithDebInfo`
- Zoek naar uw uitvoerbare bestand `eclipse.exe`.

- Maak het volgende script en gebruik dat om Eclipse te starten bij het ontwikkelen van plug-ins voor QGIS.

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

13.2 Debuggen met behulp van Eclipse en PyDev

13.2.1 Installatie

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

- Eclipse
- Aptana Eclipse Plugin of PyDev
- QGIS 2.0

13.2.2 QGIS voorbereiden

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

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

13.2.3 Eclipse instellen

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

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

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

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

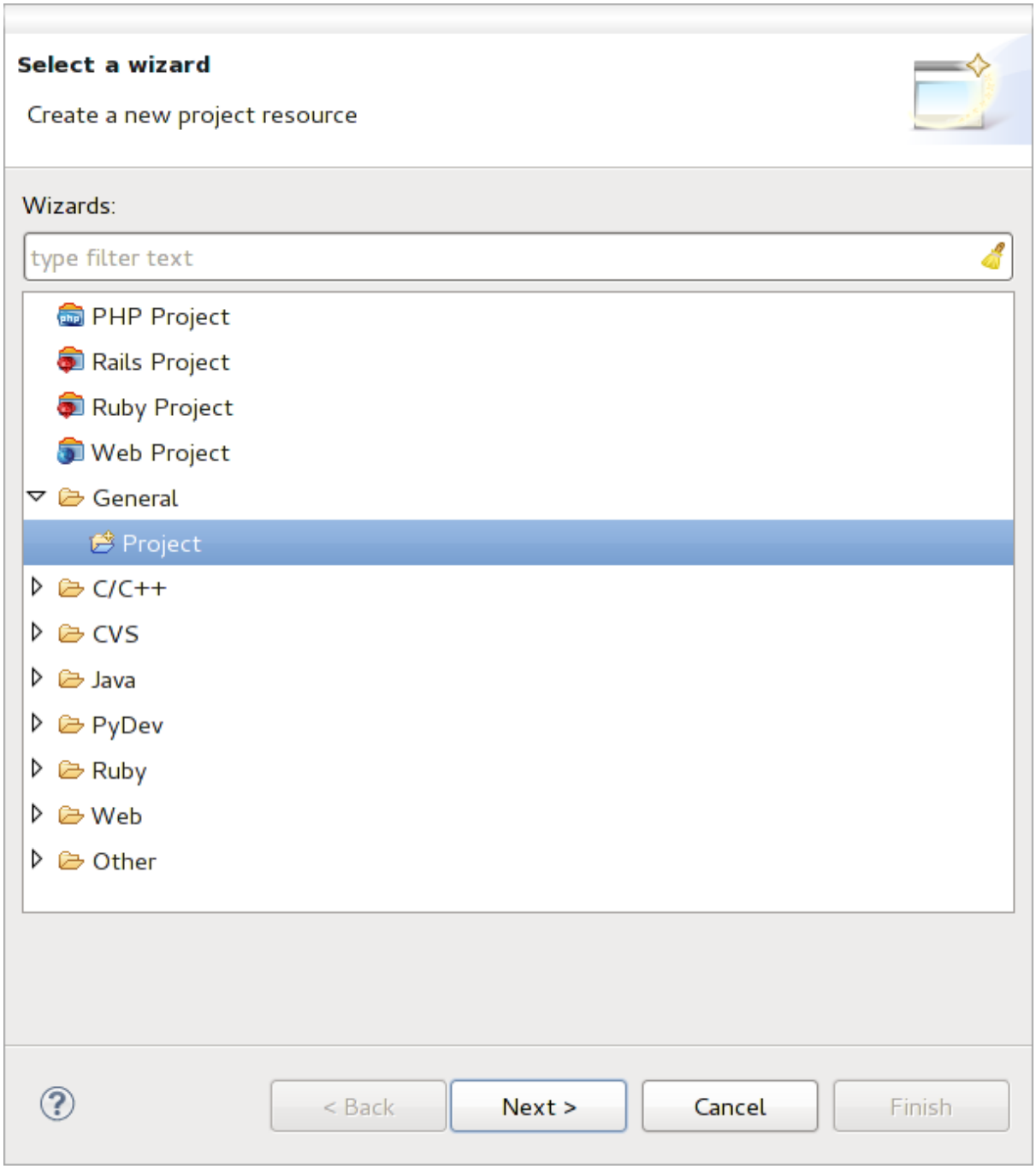
13.2.4 Configureren van de debugger

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

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

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



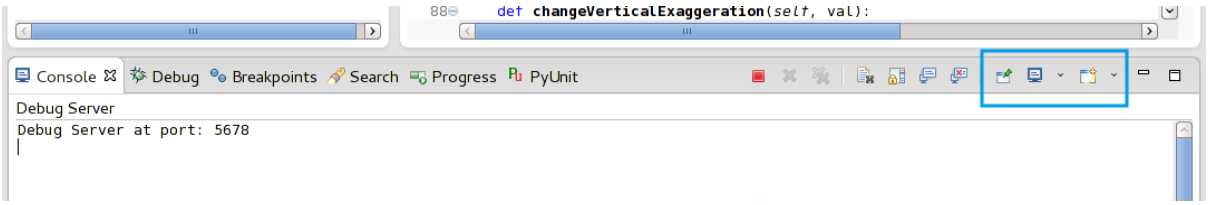
Figuur 13.1: Eclipse-project

```
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 )
```

Figuur 13.2: Onderbrekingspunt

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

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



Figuur 13.3: PyDev console voor debuggen

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

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

13.2.5 Eclipse de API laten begrijpen

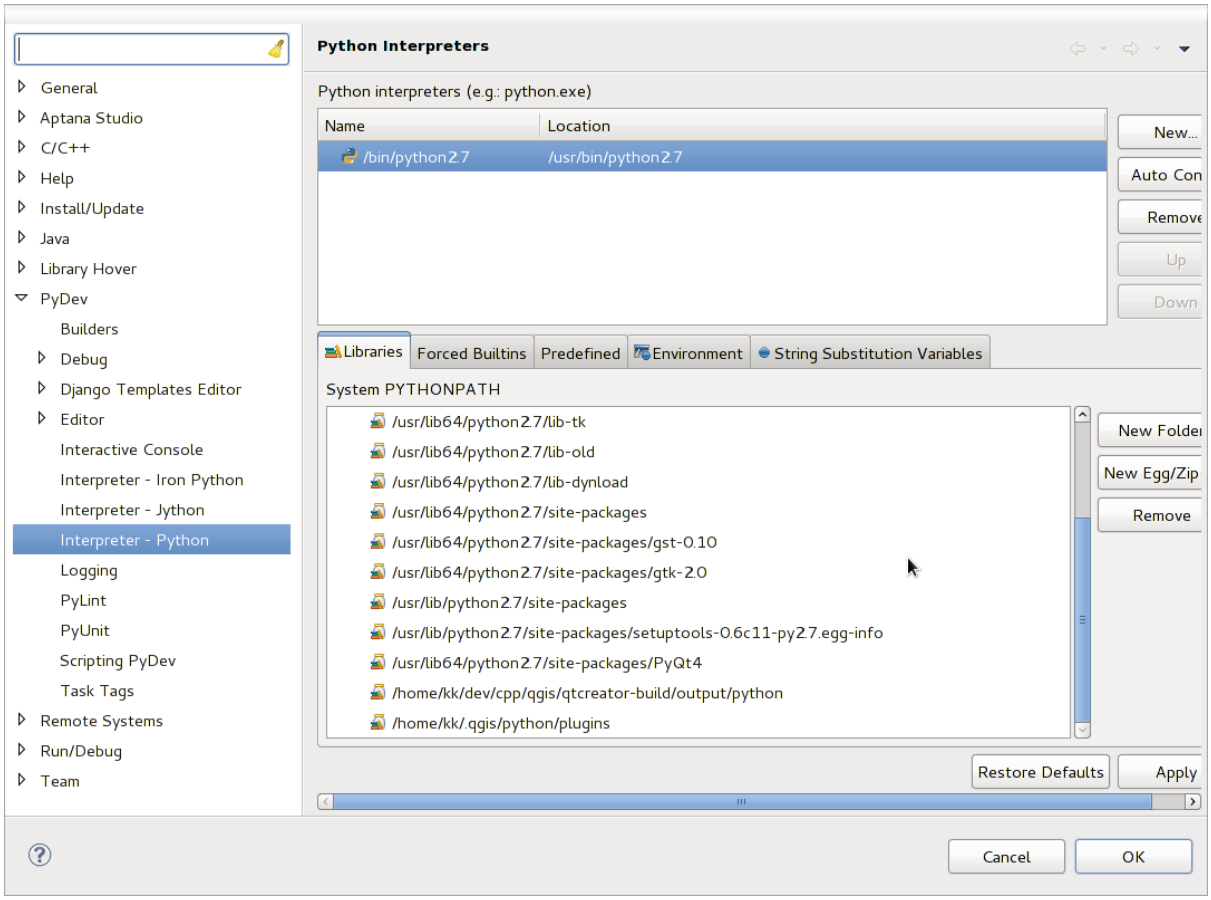
Een zeer handige mogelijkheid is om Eclipse kennis te laten nemen van de API van QGIS. Dit stelt u in staat om het uw code te laten controleren op typefouten. Maar niet alleen dat, het stelt Eclipse ook in staat om u te helpen met automatisch aanvullen vanuit het importeren naar aanroepen van de API.

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

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

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

Open erst de tab Libraries. Voeg een nieuwe map toe en kies de map voor Python van uw installatie voor QGIS. Als u niet weet waar die map staat (hij staat niet in de map plug-ins) open QGIS, start een console voor Python en



Figuur 13.4: PyDev console voor debuggen

voer eenvoudigweg `qgis` in en druk op Enter. Het zal u tonen welke module QGIS gebruikt en het pad er van. Verwijder het achterliggende `/qgis/__init__.pyc` uit dit pad en u heeft het pad waar u naar zoekt.

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

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

Klik op *OK* en u bent klaar.

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

Voor een andere mogelijke instelling van Eclipse om te werken met plug-ins voor Python in QGIS, bekijk deze link

13.3 Debuggen met behulp van PDB

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

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

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

Voer dan QGIS uit vanaf de opdrachtregel.

Doe op Linux:

$ ./Qgis

Doe op Mac OS X:

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

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

TODO: Informatie voor testen toevoegen

Plug-in-lagen gebruiken

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

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

14.1 Sub-klassen in QgsPluginLayer

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

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

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

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

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

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

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

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

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

Compatibiliteit met oudere versies van QGIS

15.1 Menu Plug-ins

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

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

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

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

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

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

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, voor voorbeelden zie andere opslagplaatsen voor plug-ins.

16.1 Officiële Python plug-in opslagplaats

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

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

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

TODO: Een koppeling naar het document voor governance invoegen

16.1.1 Rechten

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

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

16.1.2 Beheer van 'trust'

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

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

16.1.3 Validatie

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

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

1. de naam van de hoofdmap die u 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

16.1.4 Plug-in structuur

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

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

Codesnippers

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

17.1 Hoe een methode aan te roepen met een sneltoets

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

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

Aan `unload()` voeg toe

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

De methode die wordt aangeroepen wanneer op F7 wordt gedrukt

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

17.2 Hoe te schakelen tussen lagen

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

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

17.3 Hoe toegang te krijgen tot de attributentabel van geselecteerde objecten

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
    nF = layer.selectedFeatureCount()
    if (nF > 0):
    layer.startEditing()
    ob = layer.selectedFeaturesIds()
```

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

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

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

Bibliotheek Netwerkanalyse

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

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

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

18.1 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)

18.2 Een grafiek bouwen

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

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

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

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

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

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

Het is tijd om het proces in te duiken.

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

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

Dan enkele voorbeelden voor het maken van een director

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

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

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

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

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

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

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

```
properter = QgsDistanceArcProperter()
```

En de director vertellen over deze strategie

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

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

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

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

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

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

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

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

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

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

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

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

18.3 Grafiekanalyse

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

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

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

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

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

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

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

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

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

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

Hier is enige zeer eenvoudige code om de boom van het kortste pad weer te geven met behulp van de grafiek die wordt gemaakt door de methode `shortestTree()` (selecteer een lijnstring laag in de inhoudsopgave en vervang de coördinaten door uw eigen). **Waarschuwing**: gebruik deze code alleen als een voorbeeld, het maakt heel veel objecten QgsRubberBand en kan traag zijn op grote gegevenssets.

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

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

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

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

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

pStart = QgsPoint(-1.37144, 0.543836)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]
```

```
graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.arc(edgeId).inVertex()).point())
  rb.addPoint (graph.vertex(graph.arc(edgeId).outVertex()).point())
```

18.3.1 Kortste pad zoeken

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  p.append(tStart)
```

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

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

18.3.2 Beschikbare gebieden

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

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

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

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

Hier is een voorbeeld

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

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

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

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

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

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

idStart = graph.findVertex(tStart)

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

upperBound = []
r = 2000.0
i = 0
```

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

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