PyQGIS 3.40 developer cookbook

QGIS Project

aug. 16, 2025

Innehåll

KAPITEL 1

Introduktion

Detta dokument är avsett att vara både en handledning och en referensguide. Det innehåller inte en lista över alla möjliga användningsområden, men det ger en bra översikt över de viktigaste funktionerna.

Stöd för Python introducerades först i QGIS 0.9. Det finns flera sätt att använda Python i QGIS Desktop (beskrivs i följande avsnitt):

- Utföra kommandon i Python-konsolen inom QGIS
- Skapa och använda tillägg
- Kör Python-kod automatiskt när QGIS startar
- Skapa bearbetningsalgoritmer
- Skapa funktioner för uttryck i QGIS
- Skapa anpassade applikationer baserade på QGIS API

Python-bindningar finns också tillgängliga för QGIS Server, inklusive Python-tillägg (se *QGIS Server och Python*) och Python-bindningar som kan användas för att bädda in QGIS Server i en Python-applikation.

Det finns en komplett QGIS C++ API-referens som dokumenterar klasserna från QGIS-biblioteken. The Pythonic QGIS API (pyqgis) är nästan identisk med C++ API.

En annan bra resurs för att lära sig hur man utför vanliga uppgifter är att ladda ner befintliga tillägg från plugin repository och undersöka deras kod.

1.1 Skriptning i Python-konsolen

QGIS tillhandahåller en integrerad Python-konsol för skriptning. Den kan öppnas från menyn *Tillägg* ► *Python-konsol*:

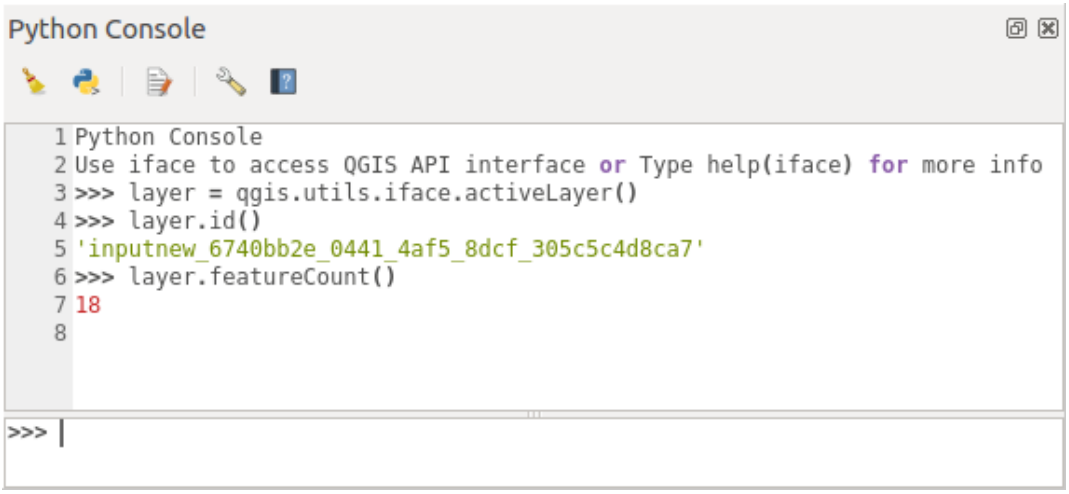


Fig. 1.1: QGIS Python-konsol

I skärmdumpen ovan illustreras hur man hämtar det lager som för närvarande är valt i lagerlistan, visar dess ID och eventuellt, om det är ett vektorlager, visar antalet funktioner. För interaktion med QGIS-miljön finns variabeln `iface`, som är en instans av `QgisInterface`. Detta gränssnitt ger tillgång till kartbilden, menyer, verktygsfält och andra delar av QGIS-programmet.

För att underlätta för användaren utförs följande kommandon när konsolen startas (i framtiden kommer det att vara möjligt att ställa in ytterligare initiala kommandon)

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

För dem som använder konsolen ofta kan det vara bra att ange en genväg för att starta konsolen (inom *Inställningar* ► *Tangentbordsgenvägar…*)

1.2 Tillägg för Python

Funktionaliteten i QGIS kan utökas med hjälp av tillägg. Tillägg kan skrivas i Python. Den största fördelen jämfört med C++-tillägg är enkel distribution (ingen kompilering för varje plattform) och enklare utveckling.

Många tillägg som täcker olika funktioner har skrivits sedan Python-stödet introducerades. Med plugin-installationsprogrammet kan användare enkelt hämta, uppgradera och ta bort Python-tillägg. Se sidan Python Tillägg för mer information om tillägg och plugin-utveckling.

Det är enkelt att skapa tillägg i Python, se *Utveckla Python-tillägg* för detaljerade instruktioner.

> **Observera**
>
> Python-tillägg finns också tillgängliga för QGIS-servern. Se *QGIS Server och Python* för ytterligare information.

1.2.1 Tillägg för bearbetning

Processing Tillägg kan användas för att bearbeta data. De är enklare att utveckla, mer specifika och mer lättviktiga än Python-tillägg. *Skriva ett Processing-tillägg* förklarar när det är lämpligt att använda Processing-algoritmer och hur man utvecklar dem.

1.3 Kör Python-kod när QGIS startar

Det finns olika metoder för att köra Python-kod varje gång QGIS startar.

1. Skapa ett skript för startup.py
2. Ställa in miljövariabeln `PYQGIS_STARTUP` till en befintlig Python-fil
3. Ange ett startskript med hjälp av parametern `--code init_qgis.py`.

1.3.1 Filen `startup.py`

Varje gång QGIS startar söks användarens Python-hjemkatalog och en lista med systemsökvägar efter en fil med namnet `startup.py`. Om den filen finns, exekveras den av den inbäddade Python-tolken.

Sökvägen i användarens hemkatalog finns vanligtvis under:

- Linux: `.local/share/QGIS/QGIS3`
- Windows: `AppData\Roaming\QGIS\QGIS3`
- macOS: `Bibliotek/Applikationsstöd/QGIS/QGIS3`

De förvalda systemsökvägarna beror på operativsystemet. För att hitta de sökvägar som fungerar för dig öppnar du Python Console och kör `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` för att se listan över standardkataloger.

Skriptet `startup.py` exekveras omedelbart efter initiering av python i QGIS, tidigt i starten av programmet.

1.3.2 Miljövariabeln PYQGIS_STARTUP

Du kan köra Python-kod precis innan QGIS-initialiseringen är klar genom att ställa in miljövariabeln `PYQGIS_STARTUP` till sökvägen för en befintlig Python-fil.

Denna kod kommer att köras innan QGIS-initialiseringen är klar. Den här metoden är mycket användbar för att rensa sys.path, som kan ha oönskade sökvägar, eller för att isolera/ladda den initiala miljön utan att kräva en virtuell miljö, t.ex. homebrew eller MacPorts-installationer på Mac.

1.3.3 Parametern ”-kod

Du kan tillhandahålla anpassad kod som ska köras som startparameter för QGIS. Det gör du genom att skapa en pythonfil, till exempel `qgis_init.py`, för att köra och starta QGIS från kommandoraden med `qgis --code qgis_init.py`.

Kod som tillhandahålls via `--code` exekveras sent i QGIS initialiseringsfas, efter att applikationskomponenterna har laddats.

1.3.4 Ytterligare argument för Python

För att ge ytterligare argument för ditt `--code`-skript eller för annan pythonkod som exekveras, kan du använda argumentet `--py-args`. Alla argument som kommer efter `--py-args` och före ett `--`-argument (om det finns) kommer att skickas till Python men ignoreras av QGIS-programmet självt.

I följande exempel kommer `myfile.tif` att vara tillgänglig via `sys.argv` i Python men kommer inte att laddas av QGIS. Medan `otherfile.tif` kommer att laddas av QGIS men finns inte i `sys.argv`.

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

Om du vill ha tillgång till alla kommandoradsparametrar från Python kan du använda `QCoreApplication.arguments()`

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

1.4 Python-tillämpningar

Det är ofta praktiskt att skapa skript för att automatisera processer. Med PyQGIS är detta fullt möjligt — importera modulen qgis.core, initiera den och du är redo för bearbetningen.

Eller så kanske du vill skapa en interaktiv applikation som använder GIS-funktionalitet — utföra mätningar, exportera en karta som PDF, … Modulen qgis.gui tillhandahåller olika grafiska gränssnittskomponenter, framför allt widgeten map canvas som kan integreras i applikationen med stöd för zoomning, panorering och/eller andra anpassade kartverktyg.

PyQGIS anpassade program eller fristående skript måste konfigureras för att lokalisera QGIS-resurserna, t.ex. projektionsinformation och providers för läsning av vektor- och rasterlager. QGIS-resurser initieras genom att lägga till några rader i början av programmet eller skriptet. Koden för att initiera QGIS för anpassade program och fristående skript är likartad. Exempel på var och en av dem ges nedan.

> **Observera**
>
> Använd *inte* `qgis.py` som namn på ditt skript. Python kommer inte att kunna importera bindningarna eftersom skriptets namn kommer att skugga dem.

1.4.1 Använda PyQGIS i fristående skript

Om du vill starta ett fristående skript initialiserar du QGIS-resurserna i början av skriptet:

```
from qgis.core import *

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

# Create a reference to the QgsApplication.  Setting the
# second argument to False disables the GUI.
qgs = QgsApplication([], False)

# Load providers
qgs.initQgis()

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

# Finally, exitQgis() is called to remove the
```

(continues on next page)

(fortsättning från föregående sida)

```
# provider and layer registries from memory
qgs.exitQgis()
```

Först importerar vi modulen qgis.core och konfigurerar prefixsökvägen. Prefixsökvägen är den plats där QGIS är installerat på ditt system. Den konfigureras i skriptet genom att anropa metoden `setPrefixPath()`. Det andra argumentet i `setPrefixPath()` är inställt på `True`, vilket anger att standardsökvägar ska användas.

Installationsstigen för QGIS varierar beroende på plattform; det enklaste sättet att hitta den för ditt system är att använda *Skriptning i Python-konsolen* inifrån QGIS och titta på utdata från körningen:

```
QgsApplication.prefixPath()
```

När prefixsökvägen har konfigurerats sparar vi en referens till `QgsApplication` i variabeln `qgs`. Det andra argumentet är inställt på `False`, vilket anger att vi inte planerar att använda det grafiska gränssnittet eftersom vi skriver ett fristående skript. När `QgsApplication` har konfigurerats laddar vi QGIS-dataleverantörerna och lagerregistret genom att anropa metoden `initQgis()`.

```
qgs.initQgis()
```

När QGIS har initialiserats är vi redo att skriva resten av skriptet. Slutligen avslutar vi genom att anropa `exitQgis()` för att ta bort dataleverantörerna och lagerregistret från minnet.

```
qgs.exitQgis()
```

1.4.2 Använda PyQGIS i egna applikationer

Den enda skillnaden mellan *Använda PyQGIS i fristående skript* och en anpassad PyQGIS-applikation är det andra argumentet vid instansiering av `QgsApplication`. Skicka `True` istället för `False` för att ange att vi planerar att använda ett grafiskt gränssnitt.

```
from qgis.core import *

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

# Create a reference to the QgsApplication.
# Setting the second argument to True enables the GUI.  We need
# this since this is a custom application.

qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

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

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

Nu kan du arbeta med QGIS API - ladda lager och göra lite bearbetning eller starta upp ett grafiskt gränssnitt med en kartduk. Möjligheterna är oändliga :-)

1.4.3 Körning av anpassade applikationer

Du måste tala om för ditt system var det ska söka efter QGIS-bibliotek och lämpliga Python-moduler om de inte finns på en välkänd plats - annars kommer Python att klaga:

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

Detta kan åtgärdas genom att ställa in miljövariabeln `PYTHONPATH`. I följande kommandon ska `<qgispath>` ersättas med din faktiska QGIS-installationssökväg:

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

Nu är sökvägen till PyQGIS-modulerna känd, men de är beroende av biblioteken `qgis_core` och `qgis_gui` (Python-modulerna fungerar bara som omslag). Sökvägen till dessa bibliotek kan vara okänd för operativsystemet, och då kommer du att få ett importfel igen (meddelandet kan variera beroende på system):

```
>>> import qgis.core
ImportError: libqgis_core.so.3.2.0: cannot open shared object file:
  No such file or directory
```

Åtgärda detta genom att lägga till de kataloger där QGIS-biblioteken finns i sökvägen för den dynamiska länkaren:

- på Linux: **`export LD_LIBRARY_PATH=/<qgispath>/lib`**
- på Windows: **`set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%`** där `<qgisrelease>` ska ersättas med den typ av utgåva du riktar dig till (t.ex. `qgis-ltr`, `qgis`, `qgis-dev`)

Dessa kommandon kan läggas in i ett bootstrap-skript som tar hand om uppstarten. När du distribuerar anpassade applikationer med PyQGIS finns det vanligtvis två möjligheter:

- kräva att användaren installerar QGIS innan han/hon installerar din applikation. Installationsprogrammet för programmet bör leta efter standardplatser för QGIS-bibliotek och låta användaren ange sökvägen om den inte hittas. Den här metoden har fördelen att den är enklare, men den kräver att användaren gör fler steg.
- paketera QGIS tillsammans med din applikation. Det kan vara svårare att släppa applikationen och paketet blir större, men användaren slipper ladda ner och installera ytterligare program.

De två driftsättningsmodellerna kan blandas. Du kan tillhandahålla en fristående applikation för Windows och macOS, men för Linux kan du låta användaren och dennes pakethanterare sköta installationen av GIS.

1.5 Tekniska anvisningar om PyQt och SIP

Vi har valt Python eftersom det är ett av de mest populära språken för skript. PyQGIS-bindningar i QGIS 3 är beroende av SIP och PyQt5. Anledningen till att SIP används i stället för det mer allmänt använda SWIG är att QGIS-koden är beroende av Qt-bibliotek. Python-bindningar för Qt (PyQt) görs med SIP och detta möjliggör sömlös integration av PyQGIS med PyQt.

KAPITEL 2

Lastning av projekt

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

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

Ibland behöver du ladda ett befintligt projekt från ett tillägg eller (oftare) när du utvecklar en fristående QGIS Python-applikation (se: *Python-tillämpningar*).

För att ladda ett projekt i den aktuella QGIS-applikationen måste du skapa en instans av klassen `QgsProject`. Det här är en singletonklass, så du måste använda dess `instance()`-metod för att göra det. Du kan anropa dess `read()`-metod genom att skicka sökvägen till det projekt som ska laddas:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt classes you will use in this script as shown below:
from qgis.core import QgsProject
# Get the project instance
project = QgsProject.instance()
# Print the current project file name (might be empty in case no projects have been loaded)
# print(project.fileName())

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

```
testdata/01_project.qgs
```

Om du behöver göra ändringar i projektet (t.ex. lägga till eller ta bort några lager) och spara dina ändringar, anropa

metoden `write()` i din projektinstans. Metoden `write()` accepterar också en valfri sökväg för att spara projektet på en ny plats:

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

Både funktionerna `read()` och `write()` returnerar ett booleskt värde som du kan använda för att kontrollera om operationen lyckades.

Observera

Om du skriver en fristående QGIS-applikation måste du instansiera en `QgsLayerTreeMapCanvasBridge` för att synkronisera det laddade projektet med canvas, som i exemplet nedan:

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

2.1 Lösning av dåliga vägar

Det kan hända att lager som laddats i projektet flyttas till en annan plats. När projektet laddas igen är alla lagerstigar trasiga. Klassen `QgsPathResolver` hjälper dig att skriva om lagersökvägen inom projektet.

Dess `setPathPreprocessor()`-metod gör det möjligt att ställa in en anpassad sökvägsförberedande funktion för att manipulera sökvägar och datakällor innan de löses till filreferenser eller lagerkällor.

Processorfunktionen måste acceptera ett enda strängargument (som representerar den ursprungliga filsökvägen eller datakällan) och returnera en bearbetad version av denna sökväg. Funktionen för sökvägspreprocessor anropas **före** alla hanterare av dåliga lager. Om flera förbehandlare är inställda kommer de att anropas i sekvens baserat på den ordning i vilken de ursprungligen ställdes in.

Några användningsområden:

1. ersätta en föråldrad väg:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

2. ersätta en databas värdadress med en ny:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. ersätta lagrade databasuppgifter med nya:

```
def my_processor(path):
    path= path.replace("user='gis_team'", "user='team_awesome'")
    path = path.replace("password='cats'", "password='g7as!m*'")
    return path

QgsPathResolver.setPathPreprocessor(my_processor)
```

På samma sätt finns en `setPathWriter()`-metod tillgänglig för en sökvägsritarfunktion.

Ett exempel på att ersätta sökvägen med en variabel:

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

QgsPathResolver.setPathWriter(my_processor)
```

Båda metoderna returnerar ett `id` som kan användas för att ta bort den förprocessor eller skrivare som de lade till. Se `removePathPreprocessor()` och `removePathWriter()`.

2.2 Använda flaggor för att påskynda saker

I vissa fall där du kanske inte behöver använda ett fullt fungerande projekt, utan bara vill komma åt det av en viss anledning, kan flaggor vara till hjälp. En fullständig lista över flaggor finns tillgänglig under `ProjectReadFlag`. Flera flaggor kan läggas till tillsammans.

Om vi t.ex. inte bryr oss om faktiska lager och data utan bara vill komma åt ett projekt (t.ex. för inställningar för layout eller 3D-vy) kan vi använda flaggan `DontResolveLayers` för att kringgå datavalideringssteget och förhindra att dialogrutan för felaktiga lager visas. Följande kan göras:

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

För att lägga till fler flaggor måste pythonoperatorn Bitwise OR (`|`) användas.

KAPITEL 3

Laddning av lager

Råd

Kodsnuttarna på den här sidan behöver följande import:

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

Låt oss öppna några lager med data. QGIS känner igen vektor- och rasterlager. Dessutom finns det anpassade lagertyper, men vi kommer inte att diskutera dem här.

3.1 Vektorlager

Om du vill skapa och lägga till en vektorlagerinstans i projektet anger du lagrets datakällsidentifierare. Datakällans identifierare är en sträng och den är specifik för varje vektordataleverantör. Ett valfritt lagernamn används för att identifiera lagret i panelen *Layers*. Det är viktigt att kontrollera om lagret har laddats på rätt sätt. Om så inte är fallet returneras en ogiltig lagerinstans.

För ett geopackage-vektorlager:

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

Det snabbaste sättet att öppna och visa ett vektorlager i QGIS är metoden `addVectorLayer()` i klassen `QgisInterface`:

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

Detta skapar ett nytt lager och lägger till det i det aktuella QGIS-projektet (vilket gör att det visas i lagerlistan) i ett steg. Funktionen returnerar lagerinstansen eller `None` om lagret inte kunde laddas.

Följande lista visar hur du får åtkomst till olika datakällor med hjälp av vektordatakällor:

- Ogr-leverantören från GDAL-biblioteket stöder en stor mängd olika format, även kallade drivrutiner i GDAL-språk. Exempel är ESRI Shapefile, Geopackage, Flatgeobuf, Geojson, … För format med en enda fil räcker det vanligtvis med filvägen som uri. För geopaket eller dxf kan ett pipseparerat suffix användas för att ange vilket lager som ska laddas.
  - för ESRI Shapefile:

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

  - för Geopackage (observera de interna alternativen i datakällans uri):

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

    Observera att om ditt GeoPackage innehåller relationer mellan lager. Att bara använda `addMapLayer` skapar inte automatiskt dessa relationer mellan lagren i ditt QGIS-projekt. Lägg till följande kod för att automatiskt hitta relationerna i databasen och lägga till lämpliga relationer i projektets egenskaper.

    ```
    from qgis.core import QgsRelationManager

    project = QgsProject.instance()
    relation_manager = project.relationManager()
    existing_relations = list(relation_manager.relations().values())

    layers = [
        layer for layer in project.mapLayers().values() if layer.type() ==␣
    ↪layer.VectorLayer
    ]

    discovered_relations = QgsRelationManager.discoverRelations(existing_
    ↪relations, layers)
    for relation in discovered_relations:
        relation_manager.addRelation(relation)
    ```

  - för dxf (notera de interna alternativen i datakällans uri):

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

- PostGIS-databas - datakälla är en sträng med all information som behövs för att skapa en anslutning till PostgreSQL-databasen.

  `QgsDataSourceUri` kan generera denna sträng åt dig. Observera att QGIS måste kompileras med stöd för Postgres, annars är den här leverantören inte tillgänglig:

  ```
  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
  ```

(continues on next page)

(fortsättning från föregående sida)

```
# subset (WHERE clause)
uri.setDataSource("public", "roads", "the_geom", "cityid = 2643", "primary_key_
↪field")

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

Observera

Argumentet `False` som skickas till `uri.uri(False)` förhindrar utvidgning av parametrarna för autentiseringskonfigurationen, om du inte använder någon autentiseringskonfiguration gör detta argument ingen skillnad.

- CSV eller andra avgränsade textfiler — för att öppna en fil med semikolon som avgränsare, med fält ”x” för X-koordinat och fält ”y” för Y-koordinat använder du något liknande detta:

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

Observera

Datakällssträngen är strukturerad som en URL, så sökvägen måste inledas med `file://`. Den tillåter också WKT (well known text)-formaterade geometrier som ett alternativ till fälten `x` och `y`, och tillåter att koordinatreferenssystemet anges. Exempelvis

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

- GPX-filer — Dataleverantören ”gpx” läser spår, rutter och waypoints från gpx-filer. För att öppna en fil måste typen (track/route/waypoint) anges som en del av webbadressen:

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

- SpatiaLite-databas — På samma sätt som PostGIS-databaser kan `QgsDataSourceUri` användas för att generera datakällans identifierare:

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

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

- MySQL WKB-baserade geometrier, genom GDAL — datakälla är anslutningssträngen till tabellen:

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

- WFS-anslutning: anslutningen definieras med en URI och med hjälp av `WFS`-datakällan:

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

URI kan skapas med hjälp av standardbiblioteket `urllib`:

```
import urllib

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

Observera

Du kan ändra datakällan för ett befintligt lager genom att anropa `setDataSource()` på en `QgsVectorLayer`-instans, som i följande exempel:

```
uri = "https://demo.mapserver.org/cgi-bin/wfs?service=WFS&version=2.0.0&
↪request=GetFeature&typename=ms:cities"
provider_options = QgsDataProvider.ProviderOptions()
# Use project's transform context
provider_options.transformContext = QgsProject.instance().transformContext()
vlayer.setDataSource(uri, "layer name you like", "WFS", provider_options)

del(vlayer)
```

3.2 Rasterlager

För åtkomst till rasterfiler används GDAL-biblioteket. Det stöder ett brett utbud av filformat. Om du har problem med att öppna vissa filer, kontrollera om din GDAL har stöd för det specifika formatet (alla format är inte tillgängliga som standard). För att ladda ett raster från en fil, ange dess filnamn och visningsnamn:

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

För att ladda ett raster från ett geopaket:

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

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

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

På samma sätt som vektorlager kan rasterlager laddas med hjälp av funktionen addRasterLayer i objektet `QgisInterface`:

```
iface.addRasterLayer(path_to_tif, "layer name you like")
```

Detta skapar ett nytt lager och lägger till det i det aktuella projektet (vilket gör att det visas i lagerlistan) i ett steg.

För att ladda ett PostGIS-raster:

PostGIS-raster kan, på samma sätt som PostGIS-vektorer, läggas till i ett projekt med hjälp av en URI-sträng. Det är effektivt att ha en återanvändbar ordbok med strängar för databasanslutningsparametrarna. Detta gör det enkelt att redigera ordlistan för den aktuella anslutningen. Ordlistan kodas sedan till en URI med hjälp av provider-metadataobjektet ”postgresraster”. Efter det kan rastret läggas till i projektet.

```
uri_config = {
    # database parameters
    'dbname':'gis_db',       # The PostgreSQL database to connect to.
    'host':'localhost',      # The host IP address or localhost.
    'port':'5432',           # The port to connect on.
    'sslmode':QgsDataSourceUri.SslDisable, # SslAllow, SslPrefer, SslRequire, SslVerifyCa, SslVerifyFull
    # user and password are not needed if stored in the authcfg or service
    'authcfg':'QconfigId',  # The QGIS athentication database ID holding connection details.
    'service': None,         # The PostgreSQL service to be used for connection to the database.
    'username':None,         # The PostgreSQL user name.
    'password':None,         # The PostgreSQL password for the user.
    # table and raster column details
    'schema':'public',       # The database schema that the table is located in.
    'table':'my_rasters',    # The database table to be loaded.
    'geometrycolumn':'rast',# raster column in PostGIS table
    'sql':None,              # An SQL WHERE clause. It should be placed at the end of the string.
    'key':None,              # A key column from the table.
    'srid':None,             # A string designating the SRID of the coordinate reference system.
    'estimatedmetadata':'False', # A boolean value telling if the metadata is estimated.
    'type':None,             # A WKT string designating the WKB Type.
    'selectatid':None,       # Set to True to disable selection by feature ID.
    'options':None,          # other PostgreSQL connection options not in this list.
    'enableTime': None,
    'temporalDefaultTime': None,
    'temporalFieldIndex': None,
    'mode':'2',              # GDAL 'mode' parameter, 2 unions raster tiles, 1 adds tiles separately (may require user input)
}
# remove any NULL parameters
uri_config = {key:val for key, val in uri_config.items() if val is not None}
# get the metadata for the raster provider and configure the URI
md = QgsProviderRegistry.instance().providerMetadata('postgresraster')
uri = QgsDataSourceUri(md.encodeUri(uri_config))

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

Rasterlager kan också skapas från en WCS-tjänst:

```
layer_name = 'modis'
url = "https://demo.mapserver.org/cgi-bin/wcs?identifier={}".format(layer_name)
rlayer = QgsRasterLayer(uri, 'my wcs layer', 'wcs')
```

Här följer en beskrivning av de parametrar som WCS-URI:n kan innehålla:

WCS URI består av **nyckel=värde**-par separerade med `&`. Det är samma format som frågesträngen i URL, kodad på samma sätt. `QgsDataSourceUri` bör användas för att konstruera URI:n för att säkerställa att specialtecken kodas korrekt.

- **url** (krävs) : URL till WCS-servern. Använd inte VERSION i URL, eftersom varje version av WCS använder olika parameternamn för **GetCapabilities**-versionen, se param version.
- **identifierare** (krävs) : Namn på täckning
- **time** (valfritt) : tidsposition eller tidsperiod (beginPosition/endPosition[/timeResolution])
- **format** (valfritt) : Namn på format som stöds. Standard är det första stödda formatet med tif i namnet eller det första stödda formatet.
- **crs** (valfritt) : CRS i form av AUTHORITY:ID, t.ex. EPSG:4326. Standard är EPSG:4326 om det stöds eller det första datoriserade bokningssystemet som stöds.
- **användarnamn** (valfritt) : Användarnamn för grundläggande autentisering.
- **lösenord** (valfritt) : Lösenord för grundläggande autentisering.
- **IgnoreGetMapUrl** (valfritt, hack) : Om angivet (satt till 1), ignorera GetCoverage URL som annonseras av GetCapabilities. Kan vara nödvändigt om en server inte är korrekt konfigurerad.
- **InvertAxisOrientation** (valfritt, hack) : Om specificerat (satt till 1), byt axel i GetCoverage-begäran. Kan vara nödvändigt för geografiska CRS om en server använder fel axelordning.
- **IgnoreAxisOrientation** (valfri, hack) : Om specificerat (satt till 1), invertera inte axelorienteringen enligt WCS-standarden för geografiska CRS.
- **cache** (valfritt) : kontroll av cache-laddning, enligt beskrivningen i QNetworkRequest::CacheLoadControl, men begäran skickas på nytt som PreferCache om den misslyckades med AlwaysCache. Tillåtna värden: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. Standardvärdet är AlwaysCache.

Alternativt kan du ladda ett rasterlager från WMS-servern. Men för närvarande är det inte möjligt att komma åt GetCapabilities-svar från API — du måste veta vilka lager du vill ha:

```
urlWithParams = "crs=EPSG:4326&format=image/png&layers=continents&styles&
↪url=https://demo.mapserver.org/cgi-bin/wms"
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print("Layer failed to load!")
```

3.3 QgsProject-instans

Om du vill använda de öppnade lagren för rendering, glöm inte att lägga till dem i `QgsProject`-instansen. Instansen `QgsProject` tar över ägandet av lagren och de kan senare nås från vilken del av applikationen som helst med hjälp av sitt unika ID. När lagret tas bort från projektet raderas det också. Lager kan tas bort av användaren i QGIS-gränssnittet, eller via Python med hjälp av metoden `removeMapLayer()`.

Att lägga till ett lager i det aktuella projektet görs med metoden `addMapLayer()`:

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

För att lägga till ett lager i en absolut position:

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

Om du vill ta bort lagret använder du metoden `removeMapLayer()`:

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

I koden ovan skickas lagrets id (du kan få det genom att anropa `id()`-metoden för lagret), men du kan också skicka själva lagerobjektet.

För en lista över laddade lager och lager-id:n, använd metoden `mapLayers()`:

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

KAPITEL 4

Tillgång till innehållsförteckningen (TOC)

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

Du kan använda olika klasser för att komma åt alla laddade lager i TOC och använda dem för att hämta information:

- `QgsProject`
- `QgsLayerTreeGroup`

4.1 Klassen QgsProject

Du kan använda `QgsProject` för att hämta information om TOC och alla lager som laddats.

Du måste skapa en `instans` av `QgsProject` och använda dess metoder för att hämta de laddade lagren.

Huvudmetoden är `mapLayers()`. Den returnerar en ordbok med de laddade lagren:

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

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

Ordbokens ”nycklar” är de unika lager-ID:n medan ”värdena” är de relaterade objekten.

Det är nu enkelt att få fram all annan information om lagren:

```
# list of layer names using list comprehension
l = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
# dictionary with key = layer name and value = layer object
```

(continues on next page)

(fortsättning från föregående sida)

```
layers_list = {}
for l in QgsProject.instance().mapLayers().values():
  layers_list[l.name()] = l

print(layers_list)
```

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

Du kan också söka i TOC med hjälp av lagrets namn:

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

Observera

En lista med alla matchande lager returneras, så vi indexerar med `[0]` för att få det första lagret med det här namnet.

4.2 Klassen QgsLayerTreeGroup

Lagerträdet är en klassisk trädstruktur som är uppbyggd av noder. Det finns för närvarande två typer av noder: gruppnoder (`QgsLayerTreeGroup`) och lagernoder (`QgsLayerTreeLayer`).

Observera

för mer information kan du läsa dessa blogginlägg av Martin Dobias: Del 1 Del 2 Del 3

Projektets lagerträd kan enkelt nås med metoden `layerTreeRoot()` i klassen `QgsProject`:

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

`root` är en gruppnod och har *barn*:

```
root.children()
```

En lista över direkta barn returneras. Undergruppsbarn bör nås från sin egen direkta förälder.

Vi kan hämta ett av barnen:

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

```
<QgsLayerTreeLayer: countries>
```

Lager kan också hämtas med hjälp av deras (unika) `id`:

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

Och grupper kan också sökas med hjälp av sina namn:

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

`QgsLayerTreeGroup` har många andra användbara metoder som kan användas för att få mer information om TOC:

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

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

Låt oss nu lägga till några lager i projektets lagerträd. Det finns två sätt att göra det på:

1. **Explicit tillägg** med hjälp av funktionerna `addLayer()` eller `insertLayer()`:

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

2. **Implicit tillägg**: eftersom projektets lagerträd är kopplat till lagerregistret räcker det med att lägga till ett lager i kartlagerregistret:

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

Du kan enkelt växla mellan `QgsVectorLayer` och `QgsLayerTreeLayer`:

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

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

Grupper kan läggas till med metoden `addGroup()`. I exemplet nedan kommer den förra att lägga till en grupp i slutet av TOC medan du för den senare kan lägga till en annan grupp inom en befintlig:

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

För att flytta noder och grupper finns det många användbara metoder.

Att flytta en befintlig nod görs i tre steg:

1. kloning av den befintliga noden
2. flytta den klonade noden till önskad position
3. radera den ursprungliga noden

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

Det är lite mer *komplicerat* att flytta runt ett lager i förklaringen:

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

(continues on next page)

(fortsättning från föregående sida)

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

eller flytta den till en befintlig grupp:

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

Några andra metoder som kan användas för att modifiera grupper och lager:

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

KAPITEL 5

Använda Raster-lager

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

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

5.1 Detaljer om lagret

Ett rasterlager består av ett eller flera rasterband — kallas single band och multi band rasters. Ett band representerar en matris av värden. En färgbild (t.ex. ett flygfoto) är ett raster som består av röda, blå och gröna band. Enbandsraster representerar vanligtvis antingen kontinuerliga variabler (t.ex. höjd) eller diskreta variabler (t.ex. markanvändning). I vissa fall levereras ett rasterlager med en palett och rastervärdena hänvisar till de färger som finns lagrade i paletten.

Följande kod förutsätter att `rlayer` är ett `QgsRasterLayer`-objekt.

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

```
919 619
```

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

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

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

```
20.0685680819999988,-34.2700107699999990 : 20.8394528430000001,-33.7507750070000014
```

```
# get the raster type: 0 = GrayOrUndefined (single band), 1 = Palette (single␣
↪band), 2 = Multiband
print(rlayer.rasterType())
```

```
0
```

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

```
1
```

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

```
Band 1: Height
```

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

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

5.2 Renderare

När ett rasterlager laddas får det en standardrenderare baserad på dess typ. Den kan ändras antingen i lagrets egenskaper eller programmatiskt.

För att fråga efter den aktuella renderingen:

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

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

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

```
singlebandgray
```

För att ställa in en rendering använder du metoden `setRenderer()` i `QgsRasterLayer`. Det finns ett antal renderingsklasser (härledda från `QgsRasterRenderer`):

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

- QgsRasterContourRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

Rasterlager med enstaka band kan ritas antingen i gråa färger (låga värden = svart, höga värden = vitt) eller med en pseudocolor-algoritm som tilldelar värdena färger. Enkelbandsraster med en palett kan också ritas med hjälp av paletten. Flerbandslager ritas vanligtvis genom att mappa banden till RGB-färger. En annan möjlighet är att bara använda ett band för ritning.

5.2.1 Raster för enstaka band

Låt oss säga att vi vill rendera ett rasterlager med ett enda band med färger som sträcker sig från grönt till gult (motsvarande pixelvärden från 0 till 255). I det första steget kommer vi att förbereda ett QgsRasterShader-objekt och konfigurera dess shader-funktion:

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

Shadern mappar färgerna enligt vad som anges i dess färgkarta. Färgkartan tillhandahålls som en lista över pixelvärden med associerade färger. Det finns tre lägen för interpolering:

- linjär (`Interpolerad`): färgen interpoleras linjärt från färgkartans poster över och under pixelvärdet
- diskret (`Discrete`): färgen hämtas från den närmaste färgkartposten med samma eller högre värde
- exakt (`Exact`): färgen interpoleras inte, endast pixlar med värden som är lika med färgkartans poster ritas

I det andra steget kommer vi att associera denna shader med rasterlagret:

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

Siffran `1` i koden ovan är bandnumret (rasterband indexeras från ett).

Slutligen måste vi använda metoden `triggerRepaint()` för att se resultatet:

```
rlayer.triggerRepaint()
```

5.2.2 Multi Band Rasters

Som standard mappar QGIS de tre första banden till rött, grönt och blått för att skapa en färgbild (detta är ritningsstilen `MultiBandColor`). I vissa fall kanske du vill åsidosätta dessa inställningar. Följande kod växlar mellan rött band (1) och grönt band (2):

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

Om endast ett band behövs för att visualisera rastret kan man välja att rita ett band, antingen i gråtoner eller pseudofärg.

Vi måste använda `triggerRepaint()` för att uppdatera kartan och se resultatet:

```
rlayer_multi.triggerRepaint()
```

5.3 Fråga värden

Rastervärden kan efterfrågas med hjälp av metoden `sample()` i klassen `QgsRasterDataProvider`. Du måste ange en `QgsPointXY` och bandnumret för det rasterlager som du vill fråga. Metoden returnerar en tupel med värdet och `True` eller `False` beroende på resultatet:

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

En annan metod för att fråga efter rastervärden är att använda metoden `identify()` som returnerar ett `QgsRasterIdentifyResult`-objekt.

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

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

```
{1: 323.0}
```

I det här fallet returnerar metoden `results()` en ordbok med bandindex som nycklar och bandvärden som värden. Till exempel något i stil med `{1: 323.0}`

5.4 Redigering av rasterdata

Du kan skapa ett rasterlager med hjälp av klassen `QgsRasterBlock`. Så här skapar du till exempel ett 2x2 rasterblock med en byte per pixel:

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

Rasterpixlar kan skrivas över tack vare metoden `writeBlock()`. För att skriva över befintliga rasterdata på position 0,0 med 2x2-blocket:

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

KAPITEL 6

Använda vektorlager

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

from qgis.core.additions.edit import edit

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

I detta avsnitt sammanfattas olika åtgärder som kan göras med vektorlager.

Det mesta arbetet här baseras på metoderna i klassen `QgsVectorLayer`.

6.1 Hämta information om attribut

Du kan hämta information om de fält som associeras med ett vektorlager genom att anropa `fields()` på ett `QgsVectorLayer`-objekt:

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

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

Metoderna `displayField()` och `mapTipTemplate()` ger information om det fält och den mall som används i fliken maptips.

När du laddar ett vektorlager väljs alltid ett fält av QGIS som `Visningsnamn`, medan `HTML Map Tip` är tomt som standard. Med dessa metoder kan du enkelt få båda:

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

```
name
```

Observera

Om du ändrar `Visningsnamn` från ett fält till ett uttryck måste du använda `displayExpression()` istället för `displayField()`.

6.2 Iteration över vektorlager

Att iterera över funktionerna i ett vektorlager är en av de vanligaste uppgifterna. Nedan följer ett exempel på en enkel grundläggande kod för att utföra denna uppgift och visa lite information om varje funktion. Variabeln `layer` antas ha ett objekt av `QgsVectorLayer`.

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

for feature in features:
```

(continues on next page)

(fortsättning från föregående sida)

```
    # retrieve every feature with its geometry and attributes
    print("Feature ID: ", feature.id())
    # fetch geometry
    # show some information about the feature geometry
    geom = feature.geometry()
    geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
    if geom.type() == QgsWkbTypes.PointGeometry:
        # the geometry type can be of single or multi type
        if geomSingleType:
            x = geom.asPoint()
            print("Point: ", x)
        else:
            x = geom.asMultiPoint()
            print("MultiPoint: ", x)
    elif geom.type() == QgsWkbTypes.LineGeometry:
        if geomSingleType:
            x = geom.asPolyline()
            print("Line: ", x, "length: ", geom.length())
        else:
            x = geom.asMultiPolyline()
            print("MultiLine: ", x, "length: ", geom.length())
    elif geom.type() == QgsWkbTypes.PolygonGeometry:
        if geomSingleType:
            x = geom.asPolygon()
            print("Polygon: ", x, "Area: ", geom.area())
        else:
            x = geom.asMultiPolygon()
            print("MultiPolygon: ", x, "Area: ", geom.area())
    else:
        print("Unknown or invalid geometry")
    # fetch attributes
    attrs = feature.attributes()
    # attrs is a list. It contains all the attribute values of this feature
    print(attrs)
    # for this test only print the first feature
    break
```

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

6.3 Välja funktioner

I QGIS desktop kan funktioner väljas på olika sätt: användaren kan klicka på en funktion, rita en rektangel på kartbilden eller använda ett uttrycksfilter. Markerade funktioner är normalt markerade i en annan färg (standard är gul) för att dra användarens uppmärksamhet till markeringen.

Ibland kan det vara användbart att programmatiskt välja funktioner eller att ändra standardfärgen.

För att välja alla funktioner kan metoden `selectAll()` användas:

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

Om du vill välja med hjälp av ett uttryck använder du metoden `selectByExpression()`:

```
# Assumes that the active layer is points.shp file from the QGIS test suite
# (Class (string) and Heading (number) are attributes in points.shp)
```

(continues on next page)

(fortsättning från föregående sida)

```
layer = iface.activeLayer()
layer.selectByExpression('"Class"=\'B52\' and "Heading" > 10 and "Heading" <70',␣
↪QgsVectorLayer.SetSelection)
```

Om du vill ändra markeringsfärgen kan du använda metoden `setSelectionColor()` i `QgsMapCanvas` enligt följande exempel:

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

Om du vill lägga till funktioner i listan över valda funktioner för ett visst lager kan du anropa `select()` och skicka listan över funktionernas ID:n till den:

```
selected_fid = []

# Get the first feature id from the layer
feature = next(layer.getFeatures())
if feature:
    selected_fid.append(feature.id())

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

För att rensa valet:

```
layer.removeSelection()
```

6.3.1 Tillgång till attribut

Attribut kan refereras till med sitt namn:

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

```
First feature
```

Alternativt kan attribut refereras till med index. Detta är lite snabbare än att använda namnet. Till exempel, för att få det andra attributet:

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

```
First feature
```

6.3.2 Iteration över utvalda funktioner

Om du bara behöver utvalda funktioner kan du använda metoden `selectedFeatures()` från vektorlagret:

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

6.3.3 Iteration över en delmängd av funktioner

Om du vill iterera över en given delmängd av funktioner i ett lager, t.ex. de som finns inom ett givet område, måste du lägga till ett `QgsFeatureRequest`-objekt till `getFeatures()`-anropet. Här är ett exempel:

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

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

För snabbhetens skull görs skärningen ofta bara med hjälp av funktionens avgränsningsbox. Det finns dock en flagga `ExactIntersect` som ser till att endast korsande funktioner returneras:

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

Med `setLimit()` kan du begränsa antalet begärda funktioner. Här är ett exempel:

```
request = QgsFeatureRequest()
request.setLimit(2)
for feature in layer.getFeatures(request):
    print(feature)
```

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

Om du behöver ett attributbaserat filter i stället för (eller utöver) ett spatialt filter som visas i exemplen ovan kan du bygga ett `QgsExpression`-objekt och skicka det till `QgsFeatureRequest`-konstruktören. Här är ett exempel:

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

Se *Uttryck, filtrering och beräkning av värden* för detaljer om den syntax som stöds av `QgsExpression`.

Begäran kan användas för att definiera de data som hämtas för varje funktion, så att iteratorn returnerar alla funktioner, men returnerar partiella data för var och en av dem.

```
# Only return selected fields to increase the "speed" of the request
request.setSubsetOfAttributes([0,2])

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

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

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

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

6.4 Modifiera vektorlager

De flesta vektordataleverantörer stöder redigering av lagerdata. Ibland stöder de bara en delmängd av möjliga redigeringsåtgärder. Använd funktionen `capabilities()` för att ta reda på vilken uppsättning funktioner som stöds.

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

```
The layer supports DeleteFeatures
```

För en lista över alla tillgängliga funktioner, se `API Documentation of QgsVectorDataProvider`.

Om du vill skriva ut en textbeskrivning av lagrets kapacitet i en kommaseparerad lista kan du använda `capabilitiesString()` som i följande exempel:

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

Genom att använda någon av följande metoder för redigering av vektorlager överförs ändringarna direkt till det underliggande datalagret (en fil, databas etc.). Om du bara vill göra tillfälliga ändringar kan du hoppa till nästa avsnitt som förklarar hur du gör *modifieringar med redigeringsbuffert*.

Observera

Om du arbetar i QGIS (antingen från konsolen eller från ett tillägg) kan det vara nödvändigt att tvinga fram en ny ritning av kartbilden för att se de ändringar du har gjort i geometrin, stilen eller attributen:

```
# If caching is enabled, a simple canvas refresh might not be sufficient
# to trigger a redraw and you must clear the cached image for the layer
if iface.mapCanvas().isCachingEnabled():
    layer.triggerRepaint()
else:
    iface.mapCanvas().refresh()
```

6.4.1 Lägg till funktioner

Skapa några `QgsFeature`-instanser och skicka en lista över dem till providern `QgsVectorDataProvider` `addFeatures()`-metoden. Den returnerar två värden: resultat (`True` eller `False`) och en lista över tillagda funktioner (deras ID anges av datalagret).

För att ställa in attributen för funktionen kan du antingen initiera funktionen genom att skicka ett `QgsFields`-objekt (du kan få det från metoden `fields()` i vektorlagret) eller anropa `initAttributes()` genom att skicka det antal fält du vill lägga till.

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.fields())
    feat.setAttributes([0, 'hello'])
    # Or set a single attribute by key or by index:
    feat.setAttribute('name', 'hello')
    feat.setAttribute(0, 'hello')
    feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(123, 456)))
    (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

6.4.2 Ta bort funktioner

För att ta bort vissa funktioner behöver du bara ange en lista med deras funktions-ID.

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

6.4.3 Modifiera funktioner

Det är möjligt att antingen ändra funktionens geometri eller att ändra vissa attribut. I följande exempel ändras först värdena för attribut med index 0 och 1 och därefter ändras objektets geometri.

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

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

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

Tips

Fördela klassen QgsVectorLayerEditUtils för redigering av enbart geometri

Om du bara behöver ändra geometrier kan du överväga att använda `QgsVectorLayerEditUtils` som tillhandahåller några användbara metoder för att redigera geometrier (översätta, infoga eller flytta vertex etc.).

6.4.4 Modifiera vektorlager med en redigeringsbuffert

När du redigerar vektorer i QGIS-programmet måste du först starta redigeringsläget för ett visst lager, sedan göra några ändringar och slutligen överföra (eller återställa) ändringarna. Alla ändringar som du gör skrivs inte förrän du överför dem - de ligger kvar i lagrets redigeringsbuffert i minnet. Det är möjligt att använda den här funktionen även programmatiskt — det är bara en annan metod för redigering av vektorlager som kompletterar den direkta användningen av dataleverantörer. Använd detta alternativ när du tillhandahåller några grafiska gränssnittsverktyg för redigering av vektorlager, eftersom det gör det möjligt för användaren att bestämma om han/hon vill göra en commit/rollback och tillåter användning av undo/redo. När ändringar bekräftas sparas alla ändringar från redigeringsbufferten till dataleverantören.

Metoderna liknar dem som vi har sett i providern, men de anropas på `QgsVectorLayer`-objektet istället.

För att dessa metoder ska fungera måste lagret vara i redigeringsläge. För att starta redigeringsläget, använd metoden `startEditing()`. För att stoppa redigeringen använder du metoderna `commitChanges()` eller `rollBack()`. Den första metoden överför alla dina ändringar till datakällan, medan den andra metoden kasserar dem och inte ändrar datakällan alls.

Om du vill ta reda på om ett lager är i redigeringsläge använder du metoden `isEditable()`.

Här har du några exempel som visar hur du kan använda dessa redigeringsmetoder.

```
from qgis.PyQt.QtCore import QMetaType

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

# add two features (QgsFeature instances)
```

(continues on next page)

(fortsättning från föregående sida)

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

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

# add new field
layer.addAttribute(QgsField("mytext", QMetaType.Type.QString))
# remove a field
layer.deleteAttribute(fieldIndex)
```

För att undo/redo ska fungera korrekt måste de ovan nämnda anropen paketeras in i undokommandon. (Om du inte bryr dig om undo/redo och vill att ändringarna ska lagras omedelbart är det enklare att *redigera med dataleverantör*.)

Så här kan du använda funktionen för att ångra:

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

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

if problem_occurred:
  layer.destroyEditCommand()
  # ... tell the user that there was a problem
  # and return

# ... more editing ...

layer.endEditCommand()
```

Metoden `beginEditCommand()` skapar ett internt ”aktivt” kommando och registrerar efterföljande ändringar i vektorlagret. Med anropet till `endEditCommand()` flyttas kommandot till ångra-stacken och användaren kommer att kunna ångra/återställa det från det grafiska gränssnittet. Om något gick fel när ändringarna gjordes kommer metoden `destroyEditCommand()` att ta bort kommandot och återställa alla ändringar som gjorts medan kommandot var aktivt.

Du kan också använda `with edit(layer)`-satsen för att packa in commit och rollback i ett mer semantiskt kodblock, som i exemplet nedan:

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

Detta kommer automatiskt att anropa `commitChanges()` i slutet. Om något undantag inträffar kommer det att `rollBack()` alla ändringar. Om ett problem uppstår inom `commitChanges()` (när metoden returnerar False) kommer ett `QgsEditError` undantag att tas upp.

6.4.5 Lägga till och ta bort fält

Om du vill lägga till fält (attribut) måste du ange en lista med fältdefinitioner. För borttagning av fält behöver du bara ange en lista med fältindex.

```
from qgis.PyQt.QtCore import QMetaType

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

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

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

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

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

När du har lagt till eller tagit bort fält i dataleverantören måste lagrets fält uppdateras eftersom ändringarna inte sprids automatiskt.

```
layer.updateFields()
```

Tips

Spara ändringar direkt med hjälp av `med` baserat kommando

Genom att använda `with edit(layer):` kommer ändringarna att överföras automatiskt genom att anropa `commitChanges()` i slutet. Om något undantag inträffar kommer det att `rollBack()` alla ändringar. Se *Modifiera vektorlager med en redigeringsbuffert*.

6.5 Använda spatialt index

Spatiala index kan dramatiskt förbättra prestandan i din kod om du behöver göra frekventa förfrågningar till ett vektorlager. Tänk dig till exempel att du skriver en interpoleringsalgoritm och att du för en given plats behöver veta de 10 närmaste punkterna från ett punktlager för att kunna använda dessa punkter för att beräkna det interpolerade värdet. Utan ett spatialt index är det enda sättet för QGIS att hitta dessa 10 punkter att beräkna avståndet från varje punkt till den angivna platsen och sedan jämföra dessa avstånd. Detta kan vara en mycket tidskrävande uppgift, särskilt om den måste upprepas för flera platser. Om det finns ett spatialt index för lagret blir operationen mycket mer effektiv.

Tänk på ett lager utan spatialt index som en telefonbok där telefonnumren inte är ordnade eller indexerade. Det enda sättet att hitta telefonnumret till en viss person är att läsa från början tills du hittar det.

Spatiala index skapas inte som standard för ett QGIS-vektorlager, men du kan skapa dem enkelt. Det här är vad du behöver göra:

- skapa ett spatialt index med hjälp av `QgsSpatialIndex`:

```
index = QgsSpatialIndex()
```

- add features to index — index tar `QgsFeature` objekt och lägger till det i den interna datastrukturen. Du kan skapa objektet manuellt eller använda ett från ett tidigare anrop till providerns `getFeatures()`-metod.

```
index.addFeature(feat)
```

- alternativt kan du ladda alla funktioner i ett lager på en gång med hjälp av bulkbelastning

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

- när det spatiala indexet har fyllts med vissa värden kan du göra några frågor

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

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

Du kan också använda det spatiala indexet `QgsSpatialIndexKDBush`. Detta index liknar det *standardiserade* `QgsSpatialIndex` men:

- stöder **endast** funktioner med en enda punkt
- är **statisk** (inga ytterligare funktioner kan läggas till i indexet efter konstruktionen)
- är **mycket snabbare!**
- möjliggör direkt hämtning av den ursprungliga funktionens punkter, utan att ytterligare funktionsförfrågningar krävs
- stöder verkliga *avståndsbaserade* sökningar, dvs. returnerar alla punkter inom en radie från en sökpunkt

6.6 Klassen QgsVectorLayerUtils

Klassen `QgsVectorLayerUtils` innehåller några mycket användbara metoder som du kan använda med vektorlager.

Till exempel förbereder metoden `createFeature()` en `QgsFeature` som ska läggas till i ett vektorlager med alla eventuella begränsningar och standardvärden för varje fält:

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

Med metoden `getValues()` kan du snabbt få fram värdena för ett fält eller uttryck:

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

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

6.7 Skapa vektorlager

Det finns flera sätt att generera en vektorskiktsdatauppsättning:

- klassen `QgsVectorFileWriter`: En praktisk klass för att skriva vektorfiler till disk, med antingen ett statiskt anrop till `writeAsVectorFormatV3()` som sparar hela vektorlagret eller skapa en instans av klassen och utfärda anrop till ärvda `addFeature()`. Den här klassen stöder alla vektorformat som GDAL stöder (GeoPackage, Shapefile, GeoJSON, KML och andra).
- klassen `QgsVectorLayer`: instansierar en dataleverantör som tolkar den angivna sökvägen (url) till datakällan för att ansluta till och komma åt data. Den kan användas för att skapa tillfälliga, minnesbaserade lager (`memory`) och ansluta till GDAL-vektordataset (`ogr`), databaser (`postgres`, `spatialite`, `mysql`, `mssql`) och mer (`wfs`, `gpx`, `delimitedtext`...).

6.7.1 Från en instans av `QgsVectorFileWriter`

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

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

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

Du kan också konvertera fält för att göra dem kompatibla med olika format genom att använda `FieldValueConverter`. Om du t.ex. vill konvertera arrayvariabeltyper (t.ex. i Postgres) till en texttyp kan du göra följande:

```
LIST_FIELD_NAME = 'xxxx'

class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

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

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

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

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

Ett destinations-CRS kan också anges — om en giltig instans av `QgsCoordinateReferenceSystem` skickas som den fjärde parametern, transformeras lagret till det CRS:et.

För giltiga drivrutinsnamn, anropa metoden `supportedFiltersAndFormats()` eller konsultera supported formats by OGR — du bör skicka värdet i kolumnen ”Code” som drivrutinsnamn.

Alternativt kan du ange om du bara vill exportera utvalda funktioner, skicka ytterligare drivrutinsspecifika alternativ för skapande eller tala om för skrivaren att inte skapa attribut… Det finns ett antal andra (valfria) parametrar; se `QgsVectorFileWriter`-dokumentationen för detaljer.

6.7.2 Direkt från funktioner

```
from qgis.PyQt.QtCore import QMetaType

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

""" create an instance of vector file writer, which will create the vector file.
Arguments:
1. path to new file (will fail if exists already)
2. field map
3. geometry type - from WKBTYPE enum
4. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem)
5. coordinate transform context
```

(continues on next page)

(fortsättning från föregående sida)

```
6. save options (driver name for the output file, encoding etc.)
"""

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

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

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

# add a feature
fet = QgsFeature()

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

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

6.7.3 Från en instans av `QgsVectorLayer`

Bland alla dataleverantörer som stöds av `QgsVectorLayer`-klassen, låt oss fokusera på de minnesbaserade lagren. Memory provider är avsedd att användas främst av tilläggs- eller tredjepartsapputvecklare. Den lagrar inte data på disk, vilket gör att utvecklare kan använda den som en snabb backend för vissa tillfälliga lager.

Providern stöder string-, int- och double-fält.

Minnesprovidern stöder även spatial indexering, som aktiveras genom att anropa providerns `createSpatialIndex()`-funktion. När det spatiala indexet har skapats kommer du att kunna iterera över funktioner inom mindre regioner snabbare (eftersom det inte är nödvändigt att traversera alla funktioner, bara de som finns i den angivna rektangeln).

En minnesprovider skapas genom att skicka `"memory"` som providersträng till `QgsVectorLayer`-konstruktören.

Konstruktorn tar också en URI som definierar geometritypen för lagret, en av följande: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, `"MultiPolygon"` eller `"None"`.

URI:n kan också ange koordinatreferenssystem, fält och indexering för minnesprovidern i URI:n. Syntaxen är:

crs=definition
: Anger koordinatreferenssystemet, där definitionen kan vara någon av de former som accepteras av `QgsCoordinateReferenceSystem.createFromString()`

index=yes
: Anger att leverantören kommer att använda ett spatialt index

field=name:type(length,precision)
: Anger ett attribut för lagret. Attributet har ett namn och eventuellt en typ (heltal, dubbel eller sträng), längd och precision. Det kan finnas flera fältdefinitioner.

Följande exempel på en URI innehåller alla dessa alternativ

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

Följande exempel på kod illustrerar hur man skapar och fyller i en minnesprovider

```
from qgis.PyQt.QtCore import QDateTime, QMetaType, Qt

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

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fmt = Qt.DateFormat.ISODate
t = QDateTime.fromString("2000-01-01T12:00:00", fmt)
fet.setAttributes(["Johnny", 2, 0.3, t])
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()
```

Låt oss slutligen kontrollera om allt gick bra

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

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

```
fields: 4
features: 1
extent: 10.0 10.0 10.0 10.0
F: 1 ['Johnny', 2, 0.3, PyQt5.QtCore.QDateTime(2000, 1, 1, 12, 0)] <QgsPointXY:␣
↪POINT(10 10)>
```

6.8 Utseende (symbologi) för vektorlager

När ett vektorlager renderas ges datans utseende av **renderer** och **symboler** som är associerade med lagret. Symboler är klasser som tar hand om ritning av visuell representation av funktioner, medan renderare bestämmer vilken symbol som ska användas för en viss funktion.

Renderingsprogrammet för ett visst lager kan erhållas enligt nedan:

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

Och med den referensen, låt oss utforska den lite

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

```
Type: singleSymbol
```

Det finns flera kända renderingstyper tillgängliga i QGIS kärnbibliotek:

Typ	Klass	Beskrivning
singleSymbol	QgsSingleSymbolRenderer	Renderar alla funktioner med samma symbol
categorizedSymt	QgsCategorizedSymbolRen	Rendering av funktioner med olika symboler för varje kategori
graduatedSymbo	QgsGraduatedSymbolRende	Renderar funktioner med en annan symbol för varje intervall av värden

Det kan också finnas några anpassade renderingstyper, så gör aldrig ett antagande att det bara finns dessa typer. Du kan fråga applikationens QgsRendererRegistry för att ta reda på vilka renderare som finns tillgängliga för närvarande:

```
print(QgsApplication.rendererRegistry().renderersList())
```

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

Det är möjligt att få en dumpning av en renderingsenhets innehåll i textform — kan vara användbart för felsökning

```
renderer.dump()
```

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

6.8.1 Renderare för en enda symbol

Du kan få symbolen som används för rendering genom att anropa `symbol()`-metoden och ändra den med `setSymbol()`-metoden (notering för C++-utvecklare: renderingsprogrammet tar äganderätten till symbolen.)

Du kan ändra symbolen som används av ett visst vektorlager genom att anropa `setSymbol()` och skicka en instans av den lämpliga symbolinstansen. Symboler för *punkt*-, *linje*- och *polygon*-lager kan skapas genom att anropa `createSimple()`-funktionen i motsvarande klasser `QgsMarkerSymbol`, `QgsLineSymbol` och `QgsFillSymbol`.

Den ordbok som skickas till `createSimple()` anger symbolens stilegenskaper.

Du kan t.ex. ersätta den symbol som används av ett visst **punkt**-lager genom att anropa `setSymbol()` och skicka en instans av en `QgsMarkerSymbol`, som i följande kodexempel:

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

`namn` anger markörens form och kan vara något av följande:

- `circle`
- `square`
- `cross`
- `rectangle`

- diamond
- pentagon
- triangle
- equilateral_triangle
- star
- regular_star
- arrow
- filled_arrowhead
- x

För att få en fullständig lista över egenskaper för det första symbollagret i en symbolinstans kan du följa exempelkoden:

```
print(layer.renderer().symbol().symbolLayers()[0].properties())
```

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

Detta kan vara användbart om du vill ändra vissa egenskaper:

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

6.8.2 Kategoriserad symbolrendering

När du använder en kategoriserad renderare kan du fråga efter och ställa in attributet som används för klassificering: använd metoderna `classAttribute()` och `setClassAttribute()`.

För att få en lista över kategorier

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

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

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

Där `value()` är värdet som används för diskriminering mellan kategorier, `label()` är en text som används för kategoribeskrivning och `symbol()`-metoden returnerar den tilldelade symbolen.

Renderaren lagrar vanligtvis även originalsymbolen och färgrampen som användes för klassificeringen: `sourceColorRamp()` och `sourceSymbol()` metoder.

6.8.3 Rendering av graderad symbol

Denna rendering är mycket lik den kategoriserade symbolrendering som beskrivs ovan, men i stället för ett attributvärde per klass arbetar den med värdeintervall och kan därför endast användas med numeriska attribut.

Om du vill veta mer om intervall som används i renderingsprogrammet

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

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

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

kan du återigen använda `classAttribute()` (för att hitta klassificeringsattributets namn), `sourceSymbol()` och `sourceColorRamp()` metoder. Dessutom finns metoden `mode()` som bestämmer hur intervallen skapades: med hjälp av lika intervall, kvantiler eller någon annan metod.

Om du vill skapa din egen graderade symbolrendering kan du göra det enligt exemplet nedan (som skapar ett enkelt tvåklassigt arrangemang)

```
from qgis.PyQt import QtGui

myVectorLayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports",
↪"Airports layer", "ogr")
myTargetField = 'scalerank'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setOpacity(myOpacity)
myRange1 = QgsRendererRange(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbol.defaultSymbol(
```

(continues on next page)

(fortsättning från föregående sida)

```
    myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setOpacity(myOpacity)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
↪"EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
```

6.8.4 Arbeta med symboler

För representation av symboler finns `QgsSymbol` basklass med tre härledda klasser:

- `QgsMarkerSymbol` — för punktfunktioner
- `QgsLineSymbol` — för linjefunktioner
- `QgsFillSymbol` — för polygonfunktioner

Varje symbol består av ett eller flera symbollager (klasser som härrör från `QgsSymbolLayer`). Symbolskikten gör den faktiska renderingen, symbolklassen i sig fungerar bara som en behållare för symbollagren.

Om du har en instans av en symbol (t.ex. från en renderare) är det möjligt att utforska den: metoden `type()` anger om det är en markör-, linje- eller fyllningssymbol. Det finns en `dump()`-metod som returnerar en kort beskrivning av symbolen. För att få en lista över symbollager:

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

```
0: SimpleMarker
```

För att ta reda på symbolens färg använd `color()` metod och `setColor()` för att ändra dess färg. Med markörsymboler kan du dessutom fråga efter symbolens storlek och rotation med metoderna `size()` och `angle()`. För linjesymboler returnerar metoden `width()` linjebredden.

Storlek och bredd är i millimeter som standard, vinklar är i grader.

Arbeta med symbollager

Som tidigare nämnts är det symbollagren (underklasser till `QgsSymbolLayer`) som bestämmer hur funktionerna ska se ut. Det finns flera grundläggande symbolskiktsklasser för allmänt bruk. Det är möjligt att implementera nya symbolskiktstyper och därmed godtyckligt anpassa hur funktioner ska återges. Metoden `layerType()` identifierar symbolskiktsklassen på ett unikt sätt — de grundläggande och standardiserade är symbolskiktstyperna `SimpleMarker`, `SimpleLine` och `SimpleFill`.

Du kan få en fullständig lista över de typer av symbollager som du kan skapa för en viss symbollagerklass med följande kod:

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

```
AnimatedMarker
EllipseMarker
FilledMarker
FontMarker
GeometryGenerator
MaskMarker
RasterMarker
SimpleMarker
SvgMarker
VectorField
```

Klassen `QgsSymbolLayerRegistry` hanterar en databas med alla tillgängliga symbolskiktstyper.

För att komma åt symbolskiktsdata använder du dess `properties()`-metod som returnerar en nyckelvärdesordbok med egenskaper som bestämmer utseendet. Varje symbolskiktstyp har en specifik uppsättning egenskaper som den använder. Dessutom finns det de generiska metoderna `color()`, `size()`, `angle()` och `width()`, med deras motsvarigheter i setter. Naturligtvis är storlek och vinkel endast tillgängliga för markörsymbolskikt och bredd för linjesymbolskikt.

Skapa egna typer av symbollager

Tänk dig att du skulle vilja anpassa hur data återges. Du kan skapa din egen symbollagerklass som ritar funktionerna exakt som du vill. Här är ett exempel på en markör som ritar röda cirklar med angiven radie

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

class FooSymbolLayer(QgsMarkerSymbolLayer):

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

  def layerType(self):
     return "FooMarker"

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

  def startRender(self, context):
    pass

  def stopRender(self, context):
      pass

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

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

Metoden `layerType()` bestämmer namnet på symbollagret; det måste vara unikt bland alla symbollager. Metoden `properties()` används för att bevara attribut. Metoden `clone()` måste returnera en kopia av symbollagret med alla attribut exakt lika. Slutligen finns det renderingsmetoder: `startRender()` anropas innan den första funktionen renderas, `stopRender()` när renderingen är klar, och `renderPoint()` anropas för att göra renderingen. Koordinaterna för punkten/punkterna är redan transformerade till utdatakoordinaterna.

För polylinjer och polygoner skulle den enda skillnaden vara i renderingsmetoden: du skulle använda `renderPolyline()` som tar emot en lista med linjer, medan `renderPolygon()` tar emot en lista med punkter på den yttre ringen som första parameter och en lista med inre ringar (eller Ingen) som andra parameter.

Vanligtvis är det bekvämt att lägga till ett grafiskt gränssnitt för att ställa in attribut för symbollagertypen så att användarna kan anpassa utseendet: i vårt exempel ovan kan vi låta användaren ställa in cirkelradie. Följande kod implementerar en sådan widget

```
from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

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

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

    def symbolLayer(self):
        return self.layer

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

Denna widget kan bäddas in i dialogrutan för symbolegenskaper. När symbollagertypen väljs i dialogrutan för symbolegenskaper skapas en instans av symbollagret och en instans av widgeten för symbollager. Sedan anropas metoden `setSymbolLayer()` för att tilldela symbollagret till widgeten. I den metoden bör widgeten uppdatera användargränssnittet för att återspegla symbollagrets attribut. Metoden `symbolLayer()` används för att hämta symbollagret igen via dialogrutan för egenskaper för att använda det för symbolen.

Vid varje ändring av attribut bör widgeten sända ut signalen `changed()` för att låta dialogrutan för egenskaper uppdatera förhandsgranskningen av symbolen.

Nu saknas bara det sista limmet: att göra QGIS medvetet om dessa nya klasser. Detta görs genom att lägga till symbollagret i registret. Det är möjligt att använda symbollagret även utan att lägga till det i registret, men vissa funktioner kommer inte att fungera: t.ex. laddning av projektfiler med de anpassade symbollagren eller oförmåga att redigera lagrets attribut i det grafiska gränssnittet.

Vi kommer att behöva skapa metadata för symbollagret

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

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

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

```

(continues on next page)

(fortsättning från föregående sida)

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

fslmetadata = FooSymbolLayerMetadata()
```

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

Du bör skicka lagertyp (samma som returneras av lagret) och symboltyp (markör/linje/fyllning) till konstruktören för den överordnade klassen. Metoden `createSymbolLayer()` tar hand om att skapa en instans av symbollagret med attribut som anges i ordlistan *props*. Och det finns metoden `createSymbolLayerWidget()` som returnerar inställningswidgeten för denna symbolskiktstyp.

Det sista steget är att lägga till detta symbollager i registret — och vi är klara.

6.8.5 Skapa anpassade renderingar

Det kan vara bra att skapa en ny renderingsimplementering om du vill anpassa reglerna för hur symboler ska väljas för rendering av funktioner. Några användningsfall där du skulle vilja göra det: symbolen bestäms utifrån en kombination av fält, storleken på symbolerna ändras beroende på aktuell skala etc.

Följande kod visar en enkel anpassad renderare som skapar två markörsymboler och slumpmässigt väljer en av dem för varje objekt

```
import random
from qgis.core import QgsWkbTypes, QgsSymbol, QgsFeatureRenderer


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

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

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

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

  def usedAttributes(self, context):
    return []

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

Konstruktören för den överordnade klassen `QgsFeatureRenderer` behöver ett namn på renderingsprogrammet (som måste vara unikt bland renderingsprogrammen). Metoden `symbolForFeature()` är den som bestämmer vilken symbol som ska användas för en viss funktion. `startRender()` och `stopRender()` tar hand om initialisering/avslutning av symbolrendering. Metoden `usedAttributes()` kan returnera en lista med fältnamn

som renderingsprogrammet förväntar sig ska finnas. Slutligen bör metoden `clone()` returnera en kopia av renderingsprogrammet.

Precis som med symbollager är det möjligt att bifoga ett grafiskt gränssnitt för konfiguration av renderingsprogrammet. Det måste härledas från `QgsRendererWidget`. Följande exempelkod skapar en knapp som gör det möjligt för användaren att ställa in den första symbolen

```
from qgis.gui import QgsRendererWidget, QgsColorButton


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

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

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

Konstruktören tar emot instanser av det aktiva lagret (`QgsVectorLayer`), den globala stilen (`QgsStyle`) och den aktuella renderingen. Om det inte finns någon renderare eller om renderaren har en annan typ kommer den att ersättas med vår nya renderare, annars kommer vi att använda den aktuella renderaren (som redan har den typ vi behöver). Widgetens innehåll bör uppdateras för att visa det aktuella läget för renderingsprogrammet. När dialogrutan för rendering accepteras anropas widgetens `renderer()`-metod för att hämta den aktuella renderingen — den kommer att tilldelas lagret.

Det sista som saknas är metadata för renderingsprogrammet och registrering i registret, annars fungerar inte laddningen av lager med renderingsprogrammet och användaren kan inte välja det från listan över renderingsprogram. Låt oss avsluta vårt RandomRenderer-exempel

```
from qgis.core import (
  QgsRendererAbstractMetadata,
  QgsRendererRegistry,
  QgsApplication
)

class RandomRendererMetadata(QgsRendererAbstractMetadata):

  def __init__(self):
    super().__init__("RandomRenderer", "Random renderer")

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

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

rrmetadata = RandomRendererMetadata()
```

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

På samma sätt som med symbollager väntar konstruktören för abstrakta metadata på namn på renderingsprogrammet, namn som är synligt för användare och eventuellt namn på renderingsprogrammets ikon. Metoden `createRenderer()` skickar en instans av `QDomElement` som kan användas för att återställa renderarens tillstånd från DOM-trädet. Metoden `createRendererWidget()` skapar konfigurationswidgeten. Den behöver inte vara närvarande eller kan returnera `None` om renderaren inte har något grafiskt gränssnitt.

För att associera en ikon med renderingsprogrammet kan du tilldela den i `QgsRendererAbstractMetadata`-konstruktören som ett tredje (valfritt) argument — basklassens konstruktör i RandomRendererMetadata `__init__()`-funktionen blir

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

Ikonen kan också associeras vid ett senare tillfälle med hjälp av `setIcon()`-metoden i metadataklassen. Ikonen kan laddas från en fil (som visas ovan) eller laddas från en Qt-resurs (PyQt5 innehåller .qrc-kompilatorn för Python).

6.9 Ytterligare ämnen

TODO:

- skapa/ändra symboler
- arbeta med stil (`QgsStyle`)
- arbeta med färgramper (`QgsColorRamp`)
- utforska symbollager och renderingsregister

KAPITEL 7

Geometrihantering

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

Punkter, linjestrengar och polygoner som representerar en spatial egenskap kallas vanligen geometrier. I QGIS representeras de med klassen `QgsGeometry`.

Ibland är en geometri i själva verket en samling enkla (endelade) geometrier. En sådan geometri kallas en flerdelsgeometri. Om den bara innehåller en typ av enkel geometri kallar vi den multipunkt, multilinjestring eller multipolygon. Till exempel kan ett land som består av flera öar representeras som en multipolygon.

Geometriernas koordinater kan vara i valfritt koordinatreferenssystem (CRS). När funktioner hämtas från ett lager kommer associerade geometrier att ha koordinater i lagrets CRS.

Beskrivning och specifikationer av alla möjliga geometrier, konstruktioner och relationer finns tillgängliga i OGC Simple Feature Access Standards för avancerade detaljer.

7.1 Geometri Konstruktion

PyQGIS erbjuder flera alternativ för att skapa en geometri:

- från koordinater

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

Koordinater anges med hjälp av klassen `QgsPoint` eller `QgsPointXY`. Skillnaden mellan dessa klasser är att `QgsPoint` stöder M- och Z-dimensioner.

En Polyline (Linjestring) representeras av en lista med punkter.

En polygon representeras av en lista med linjära ringar (dvs. slutna linjestringar). Den första ringen är den yttre ringen (gränsen), valfria efterföljande ringar är hål i polygonen. Observera att till skillnad från vissa program kommer QGIS att stänga ringen åt dig så att du inte behöver duplicera den första punkten som den sista.

Geometrier med flera delar går en nivå längre: multi-point är en lista med punkter, multi-linestring är en lista med linestrings och multi-polygon är en lista med polygoner.

- från välkänd text (WKT)

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

- från välkänd binär (WKB)

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

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

7.2 Tillgång till geometri

Först bör du ta reda på geometritypen. Metoden `wkbType()` är den som ska användas. Den returnerar ett värde från `QgsWkbTypes.Type` uppräkning.

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

Som ett alternativ kan man använda metoden `type()` som returnerar ett värde från uppräkningen `QgsWkbTypes.GeometryType`.

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

Du kan använda funktionen `displayString()` för att få en geometrityp som är läsbar för människor.

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

Det finns också en hjälpfunktion `isMultipart()` för att ta reda på om en geometri är multipart eller inte.

För att extrahera information från en geometri finns det accessorfunktioner för varje vektortyp. Här är ett exempel på hur man använder dessa accessorer:

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

Observera

Tuplerna (x,y) är inte riktiga tupler, de är `QgsPoint`-objekt, värdena är tillgängliga med `x()` och `y()`-metoder.

För geometrier med flera delar finns liknande accessorfunktioner: `asMultiPoint()`, `asMultiPolyline()` och `asMultiPolygon()`.

Det är möjligt att iterera över alla delar av en geometri, oavsett geometrins typ. T.ex.

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

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

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

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

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

```
Point (11 12)
```

Det är också möjligt att modifiera varje del av geometrin med `QgsGeometry.parts()`-metoden.

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

(continues on next page)

(fortsättning från föregående sida)

```
  )

print(geom.asWkt())
```

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

7.3 Geometri Predikat och operationer

QGIS använder GEOS-biblioteket för avancerade geometrioperationer som geometripredikater (`contains()`, `intersects()`, …) och set-operationer (`combine()`, `difference()`, …). Den kan också beräkna geometriska egenskaper för geometrier, t.ex. area (för polygoner) eller längder (för polygoner och linjer).

Låt oss se ett exempel som kombinerar iterering över funktionerna i ett visst lager och utförande av några geometriska beräkningar baserat på deras geometrier. Nedanstående kod kommer att beräkna och skriva ut arean och omkretsen för varje land i lagret `countries` i vårt QGIS-projekt för handledning.

Följande kod förutsätter att `layer` är ett `QgsVectorLayer`-objekt som har funktionstypen Polygon.

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

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

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

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

Nu har du beräknat och skrivit ut geometriernas area och omkrets. Du kanske dock snabbt märker att värdena är konstiga. Det beror på att areor och omkretsar inte tar hänsyn till CRS när de beräknas med hjälp av metoderna `area()` och `length()` från klassen `QgsGeometry`. För en mer kraftfull beräkning av yta och avstånd kan klassen `QgsDistanceArea` användas, som kan utföra ellipsoidbaserade beräkningar:

Följande kod förutsätter att `layer` är ett `QgsVectorLayer`-objekt som har funktionstypen Polygon.

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

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

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

```

(continues on next page)

(fortsättning från föregående sida)

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

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

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

Alternativt kanske du vill veta avståndet mellan två punkter.

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

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

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

Du kan hitta många exempel på algoritmer som ingår i QGIS och använda dessa metoder för att analysera och omvandla vektordata. Här är några länkar till koden för några av dem.

- Avstånd och area med hjälp av `QgsDistanceArea` class: Distance matrix algorithm
- Algoritm för linjer till polygoner

KAPITEL 8

Stöd för projektioner

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

8.1 Referenssystem för koordinater

Koordinatreferenssystem (CRS) är inkapslade av `QgsCoordinateReferenceSystem`-klassen. Instanser av denna klass kan skapas på flera olika sätt:

- ange CRS med dess ID

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

```
True
```

QGIS stöder olika CRS-identifierare med följande format:

  - `EPSG:<code>` — ID tilldelat av EPSG-organisationen - hanteras med `createFromOgcWms()`
  - `POSTGIS:<srid>`— ID som används i PostGIS-databaser - hanteras med `createFromSrid()`
  - `INTERNAL:<srsid>` — ID som används i den interna QGIS-databasen - hanteras med `createFromSrsId()`
  - `PROJ:<proj>` - hanterad med `createFromProj()`
  - `WKT:<wkt>` - hanterad med `createFromWkt()`

Om inget prefix anges antas WKT-definitionen.

- ange CRS genom dess välkända text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.
↪257223563]],' \
      'PRIMEM["Greenwich", 0.0], UNIT["degree",0.01745329251994295],' \
      'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
print(crs.isValid())
```

```
True
```

- skapa en ogiltig CRS och sedan använda en av funktionerna `create*` för att initialisera den. I följande exempel använder vi en Proj-sträng för att initiera projektionen.

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

```
True
```

Det är klokt att kontrollera om skapandet (d.v.s. uppslagningen i databasen) av CRS har lyckats: `isValid()` måste returnera `True`.

Observera att för initialisering av spatiala referenssystem måste QGIS leta upp lämpliga värden i sin interna databas `srs.db`. Om du skapar en oberoende applikation måste du därför ställa in sökvägar korrekt med `QgsApplication.setPrefixPath()`, annars kommer den inte att hitta databasen. Om du kör kommandona från QGIS Python-konsolen eller utvecklar ett tillägg behöver du inte bry dig: allt är redan förberett åt dig.

Tillgång till information om spatialt referenssystem:

```
crs = QgsCoordinateReferenceSystem("EPSG:4326")

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

Utdata:

```
QGIS CRS ID: 3452
PostGIS SRID: 4326
Description: WGS 84
Projection Acronym: longlat
Ellipsoid Acronym: EPSG:7030
Proj String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

8.2 CRS-omvandling

Du kan göra transformationer mellan olika spatiala referenssystem genom att använda klassen `QgsCoordinateTransform`. Det enklaste sättet att använda den är att skapa ett CRS för källa och destination och konstruera en `QgsCoordinateTransform`-instans med dem och det aktuella projektet. Sedan anropar du bara upprepade gånger `transform()`-funktionen för att göra transformationen. Som standard gör den framåtriktad transformation, men den kan också göra invers transformation.

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

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

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

Utdata:

```
Transformed point: <QgsPointXY: POINT(832713.79873844375833869 553423.
↪98688333143945783)>
Transformed back: <QgsPointXY: POINT(18 4.9999999999999911)>
```

KAPITEL 9

Använda Map Canvas

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

from qgis.PyQt.QtCore import Qt, QRectF

from qgis.PyQt.QtWidgets import QMenu

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

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

Widgeten Map canvas är förmodligen den viktigaste widgeten i QGIS eftersom den visar kartan sammansatt av överlagrade kartlager och tillåter interaktion med kartan och lagren. Kanvasen visar alltid en del av kartan som definieras av den aktuella kanvasträckningen. Interaktionen sker genom användning av **kartverktyg**: det finns verktyg för panorering, zoomning, identifiering av lager, mätning, vektorredigering och annat. I likhet med andra grafikprogram är det alltid ett verktyg som är aktivt och användaren kan växla mellan de tillgängliga verktygen.

Map Canvas implementeras med klassen `QgsMapCanvas` i modulen qgis.gui. Implementeringen är baserad på Qt

Graphics View-ramverket. Detta ramverk tillhandahåller i allmänhet en yta och en vy där anpassade grafiska objekt placeras och användaren kan interagera med dem. Vi antar att du är tillräckligt bekant med Qt för att förstå begreppen grafisk scen, vy och objekt. Om inte, läs gärna översikten över ramverket.

När kartan har panorerats, zoomats in/ut (eller någon annan åtgärd som utlöser en uppdatering), renderas kartan igen inom den aktuella omfattningen. Lagren renderas till en bild (med hjälp av klassen `QgsMapRendererJob`) och den bilden visas på duken. Klassen `QgsMapCanvas` kontrollerar också uppfräschning av den återgivna kartan. Förutom det här objektet som fungerar som bakgrund kan det finnas fler **map canvas-objekt**.

Typiska objekt i kartans canvas är gummiband (används för mätning, vektorredigering etc.) eller vertexmarkörer. Canvasobjekten används vanligtvis för att ge visuell feedback till kartverktyg, t.ex. när en ny polygon skapas skapar kartverktyget ett canvasobjekt med gummiband som visar polygonens aktuella form. Alla kartans canvasobjekt är underklasser till `QgsMapCanvasItem` som lägger till lite mer funktionalitet till de grundläggande `QGraphicsItem`-objekten.

Sammanfattningsvis består Map Canvas-arkitekturen av tre koncept:

- kartduk — för visning av kartan
- map canvas items — ytterligare objekt som kan visas på kartbilden
- kartverktyg — för interaktion med kartbilden

9.1 Inbäddning av Map Canvas

Map canvas är en widget som alla andra Qt-widgetar, så det är lika enkelt att använda den som att skapa och visa den.

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

Detta skapar ett fristående fönster med en kartbild. Det kan också bäddas in i en befintlig widget eller ett fönster. När du använder `.ui`-filer och Qt Designer, placera en `QWidget` på formuläret och befordra den till en ny klass: ange `QgsMapCanvas` som klassnamn och ange `qgis.gui` som header-fil. Verktyget `pyuic5` kommer att ta hand om det. Detta är ett mycket bekvämt sätt att bädda in canvas. Den andra möjligheten är att manuellt skriva koden för att konstruera map canvas och andra widgets (som barn till ett huvudfönster eller en dialogruta) och skapa en layout.

Som standard har Map Canvas svart bakgrund och använder inte anti-aliasing. Så här ställer du in vit bakgrund och aktiverar anti-aliasing för smidig rendering

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

(Om du undrar så kommer `Qt` från modulen `PyQt.QtCore` och `Qt.white` är en av de fördefinierade `QColor`-instanserna)

Nu är det dags att lägga till några kartlager. Vi öppnar först ett lager och lägger till det i det aktuella projektet. Sedan ställer vi in canvasens utsträckning och listan med lager för canvasen.

```
1  vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports␣
   ↪layer", "ogr")
2  if not vlayer.isValid():
3      print("Layer failed to load!")
4
5  # add layer to the registry
6  QgsProject.instance().addMapLayer(vlayer)
7
8  # set extent to the extent of our layer
9  canvas.setExtent(vlayer.extent())
10
11 # set the map canvas layer set
12 canvas.setLayers([vlayer])
```

När du har utfört dessa kommandon ska duken visa det lager som du har laddat.

9.2 Gummiband och spetsmarkörer

För att visa ytterligare data ovanpå kartan i canvas, använd map canvas-objekt. Det är möjligt att skapa egna klasser för canvasobjekt (beskrivs nedan), men det finns två användbara klasser för canvasobjekt för bekvämlighetens skull: `QgsRubberBand` för att rita polylinjer eller polygoner, och `QgsVertexMarker` för att rita punkter. De arbetar båda med kartkoordinater, så formen flyttas/skalas automatiskt när duken panoreras eller zoomas.

För att visa en polylinje:

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

För att visa en polygon

```
r = QgsRubberBand(canvas, QgsWkbTypes.PolygonGeometry)  # polygon
points = [[QgsPointXY(-100, 35), QgsPointXY(10, 50), QgsPointXY(120, 35)]]
r.setToGeometry(QgsGeometry.fromPolygonXY(points), None)
```

Observera att points for polygon inte är en vanlig lista: i själva verket är det en lista med ringar som innehåller linjära ringar av polygonen: den första ringen är den yttre gränsen, ytterligare (valfria) ringar motsvarar hål i polygonen.

Gummiband tillåter viss anpassning, nämligen att ändra färg och linjebredd

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

Canvas-objekten är bundna till canvas-scenen. För att tillfälligt dölja dem (och visa dem igen) använder du kombinationerna `hide()` och `show()`. För att helt ta bort objektet måste du ta bort det från canvasens scen

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

(i C++ är det möjligt att bara ta bort objektet, men i Python skulle `del r` bara ta bort referensen och objektet kommer fortfarande att existera eftersom det ägs av canvas)

Gummiband kan också användas för att rita punkter, men klassen `QgsVertexMarker` är bättre lämpad för detta (`QgsRubberBand` skulle bara rita en rektangel runt den önskade punkten).

Du kan använda vertexmarkören så här:

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

Detta kommer att rita ett rött kors på position **[10,45]**. Det är möjligt att anpassa ikontyp, storlek, färg och pennbredd

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

Använd samma metoder som för gummiband för att tillfälligt dölja vertexmarkörer och ta bort dem från duken.

9.3 Använda kartverktyg med Canvas

I följande exempel konstrueras ett fönster som innehåller en kartduk och grundläggande kartverktyg för panorering och zoomning av kartan. Åtgärder skapas för aktivering av varje verktyg: panorering görs med `QgsMapToolPan`, zoomning in/ut med ett par `QgsMapToolZoom`-instanser. Åtgärderna är inställda som kontrollerbara och tilldelas senare verktygen för att möjliggöra automatisk hantering av åtgärdernas markerade/okryssade tillstånd – när ett kartverktyg aktiveras markeras dess åtgärd som vald och åtgärden för det föregående kartverktyget avmarkeras. Kartverktygen aktiveras med hjälp av `setMapTool()`-metoden.

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

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

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

        self.canvas.setExtent(layer.extent())
        self.canvas.setLayers([layer])

        self.setCentralWidget(self.canvas)

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

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

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

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

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

        self.pan()

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

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

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

Du kan prova ovanstående kod i Python-konsolredigeraren. För att anropa canvasfönstret, lägg till följande rader för

att instansiera klassen `MyWnd`. De kommer att återge det för närvarande valda lagret på den nyskapade duken

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

9.3.1 Välj en funktion med hjälp av QgsMapToolIdentifyFeature

Du kan använda kartverktyget `QgsMapToolIdentifyFeature` för att be användaren att välja en funktion som skickas till en återuppringningsfunktion.

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

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

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

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

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

9.3.2 Lägg till objekt i den kontextuella menyn i kartbilden

Interaktion med kartbilden kan också ske genom poster som du kan lägga till i dess kontextuella meny med hjälp av signalen `contextMenuAboutToShow`.

Följande kod lägger till *My menu* ► *My Action* action bredvid standardposter när du högerklickar över kartbilden.

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

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

9.4 Skriva anpassade kartverktyg

Du kan skriva dina egna verktyg för att implementera ett anpassat beteende för åtgärder som utförs av användare på Canvas.

Kartverktyg bör ärva från `QgsMapTool`, klassen eller någon härledd klass, och väljas som aktiva verktyg i canvas med `setMapTool()`-metoden som vi redan har sett.

Här är ett exempel på ett kartverktyg som gör det möjligt att definiera en rektangulär utsträckning genom att klicka och dra på duken. När rektangeln är definierad skrivs dess gränskoordinater ut i konsolen. Det använder de gummibandselement som beskrivs tidigare för att visa den valda rektangeln medan den definieras.

```
class RectangleMapTool(QgsMapToolEmitPoint):
  def __init__(self, canvas):
    self.canvas = canvas
    QgsMapToolEmitPoint.__init__(self, self.canvas)
    self.rubberBand = QgsRubberBand(self.canvas, QgsWkbTypes.PolygonGeometry)
    self.rubberBand.setColor(Qt.red)
    self.rubberBand.setWidth(1)
    self.reset()

  def reset(self):
    self.startPoint = self.endPoint = None
    self.isEmittingPoint = False
    self.rubberBand.reset(QgsWkbTypes.PolygonGeometry)

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

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

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

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

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

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

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

(continues on next page)

(fortsättning från föregående sida)

```
63      self.deactivated.emit()
```

9.5 Skriva anpassade Map Canvas-objekt

Här är ett exempel på ett anpassat canvasobjekt som ritar en cirkel:

```
1  class CircleCanvasItem(QgsMapCanvasItem):
2    def __init__(self, canvas):
3      super().__init__(canvas)
4      self.center = QgsPoint(0, 0)
5      self.size   = 100
6
7    def setCenter(self, center):
8      self.center = center
9
10   def center(self):
11     return self.center
12
13   def setSize(self, size):
14     self.size = size
15
16   def size(self):
17     return self.size
18
19   def boundingRect(self):
20     return QRectF(self.center.x() - self.size/2,
21       self.center.y() - self.size/2,
22       self.center.x() + self.size/2,
23       self.center.y() + self.size/2)
24
25   def paint(self, painter, option, widget):
26     path = QPainterPath()
27     path.moveTo(self.center.x(), self.center.y());
28     path.arcTo(self.boundingRect(), 0.0, 360.0)
29     painter.fillPath(path, QColor("red"))
30
31
32 # Using the custom item:
33 item = CircleCanvasItem(iface.mapCanvas())
34 item.setCenter(QgsPointXY(200,200))
35 item.setSize(80)
```

KAPITEL 10

Rendering och utskrift av kartor

Råd

Kodsnuttarna på den här sidan behöver följande import:

```
import os

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

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

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

Det finns i allmänhet två tillvägagångssätt när indata ska återges som en karta: antingen gör du det snabbt med hjälp av *QgsMapRendererJob* eller så producerar du mer finjusterade utdata genom att komponera kartan med `QgsLayout`-klassen.

10.1 Enkel rendering

Renderingen görs genom att skapa ett `QgsMapSettings`-objekt för att definiera renderingsinställningarna och sedan konstruera ett `QgsMapRendererJob` med dessa inställningar. Den senare används sedan för att skapa den resulterande bilden.

Här är ett exempel:

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

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

render = QgsMapRendererParallelJob(settings)

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

render.finished.connect(finished)

# Start the rendering
render.start()

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

10.2 Rendering av lager med olika CRS

Om du har mer än ett lager och de har olika CRS fungerar det enkla exemplet ovan förmodligen inte: för att få rätt värden från omfattningsberäkningarna måste du uttryckligen ange destinationens CRS

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

10.3 Utskrift med hjälp av utskriftslayout

Print layout är ett mycket praktiskt verktyg om du vill göra en mer sofistikerad utskrift än den enkla rendering som visas ovan. Det är möjligt att skapa komplexa kartlayouter som består av kartvyer, etiketter, teckenförklaringar, tabeller och andra element som vanligtvis finns på papperskartor. Layouterna kan sedan exporteras till PDF, SVG, rasterbilder eller skrivas ut direkt på en skrivare.

Layouten består av ett antal klasser. De tillhör alla kärnbiblioteket. QGIS-applikationen har ett bekvämt grafiskt gränssnitt för placering av elementen, även om det inte finns tillgängligt i det grafiska gränssnittsbiblioteket. Om du inte är bekant med Qt Graphics View framework, uppmuntras du att kontrollera dokumentationen nu, eftersom layouten är baserad på den.

Den centrala klassen i layouten är `QgsLayout`, som härstammar från Qt QGraphicsScene. Låt oss skapa en instans av den:

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

Detta initierar layouten med vissa standardinställningar, särskilt genom att lägga till en tom A4-sida i layouten. Du kan skapa layouter utan att anropa `initializeDefaults()`-metoden, men du måste själv ta hand om att lägga till sidor i layouten.

Den föregående koden skapar en ”tillfällig” layout som inte är synlig i det grafiska gränssnittet. Det kan vara praktiskt att t.ex. snabbt lägga till några objekt och exportera utan att ändra själva projektet eller exponera dessa ändringar för användaren. Om du vill att layouten ska sparas/återställas tillsammans med projektet och vara tillgänglig i layouthanteraren, lägg då till:

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

Nu kan vi lägga till olika element (karta, etikett, …) i layouten. Alla dessa objekt representeras av klasser som ärver från basklassen `QgsLayoutItem`.

Här följer en beskrivning av några av de viktigaste layoutobjekten som kan läggas till i en layout.

- map — Här skapar vi en karta i en anpassad storlek och återger den aktuella kartbilden

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

- label — gör det möjligt att visa etiketter. Det är möjligt att ändra dess teckensnitt, färg, justering och marginal

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

- legend

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

- skalstreck

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

- nodbaserad form

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

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

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

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

När ett objekt har lagts till i layouten kan det flyttas och ändra storlek:

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

Som standard ritas en ram runt varje objekt. Du kan ta bort den på följande sätt:

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

Förutom att skapa layoutobjekten för hand har QGIS stöd för layoutmallar som i princip är kompositioner med alla deras objekt sparade i en .qpt-fil (med XML-syntax).

När kompositionen är klar (layoutobjekten har skapats och lagts till i kompositionen) kan vi fortsätta med att producera en raster- och/eller vektorutskrift.

10.3.1 Kontroll av layoutens giltighet

En layout består av en uppsättning sammankopplade objekt och det kan hända att dessa kopplingar bryts under ändringar (en förklaring kopplad till en borttagen karta, ett bildobjekt med saknad källfil, …) eller så kanske du vill tillämpa anpassade begränsningar på layoutobjekten. `QgsAbstractValidityCheck` hjälper dig att uppnå detta.

En grundläggande kontroll ser ut som följer:

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

Här är en kontroll som ger en varning när ett layoutkartobjekt är inställt på webbmerkuratorprojektionen:

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

  return results
```

Och här är ett mer komplext exempel, som ger en varning om något av kartobjekten i layouten är inställt på en CRS som endast är giltig utanför den utsträckning som visas i det kartobjektet:

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

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

    return results
```

10.3.2 Exportera layouten

För att exportera en layout måste klassen `QgsLayoutExporter` användas.

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

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

Använd `exportToSvg()` eller `exportToImage()` om du vill exportera till en SVG- eller bildfil istället för en PDF-fil.

10.3.3 Exportera en layoutatlas

Om du vill exportera alla sidor från en layout som har atlasalternativet konfigurerat och aktiverat måste du använda metoden `atlas()` i exportören (`QgsLayoutExporter`) med små justeringar. I följande exempel exporteras sidorna till PNG-bilder:

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

Observera att utdata sparas i mappen Base Path med hjälp av det uttryck för utdatafilnamn som konfigurerats på Atlas.

KAPITEL 11

Uttryck, filtrering och beräkning av värden

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

QGIS har visst stöd för parsning av SQL-liknande uttryck. Endast en liten delmängd av SQL-syntaxen stöds. Uttrycken kan utvärderas antingen som booleska predikat (returnerar `True` eller `False`) eller som funktioner (returnerar ett skalärt värde). Se vector_expressions i användarhandboken för en fullständig lista över tillgängliga funktioner.

Tre grundläggande typer stöds:

- tal — både heltal och decimaltal, t.ex. `123`, `3,14`
- sträng — måste de omslutas av enkla citattecken: `'hej världen'`
- kolumnreferens — vid utvärdering ersätts referensen med fältets faktiska värde. Namnen är inte escapade.

Följande funktioner är tillgängliga:

- aritmetiska operatorer: `+`, `-`, `*`, `/`, `^`
- parenteser: för att förstärka operatörens företräde: `(1 + 1) * 3`
- unär plus och minus: `-12`, `+5`
- matematiska funktioner: `qrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`

- omvandlingsfunktioner: `to_int`, `to_real`, `to_string`, `to_date`
- geometrifunktioner: `$area`, `$längd`
- funktioner för geometrihantering: `$x`, `$y`, `$geometri`, `num_geometrier`, `centroid`

Och följande predikat stöds:

- jämförelse: =, !=, >, >=, <, <=
- mönstermatchning: `LIKE` (använder % and _), ~ (reguljära uttryck)
- logiska predikat: `OCH`, `ELLER`, `INTE`
- Kontroll av NULL-värde: `IS NULL`, `IS NOT NULL`

Exempel på predikat:

- `1 + 2 = 3`
- `sin(vinkel) > 0`
- `'Hello' LIKE 'He%'`
- `(x > 10 OCH y > 10) ELLER z = 0`

Exempel på skalära uttryck:

- `2 ^ 10`
- `sqrt(val)`
- `$längd + 1`

11.1 Parsning av uttryck

I följande exempel visas hur man kontrollerar om ett visst uttryck kan analyseras korrekt:

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

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

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

11.2 Utvärdering av uttryck

Uttryck kan användas i olika sammanhang, t.ex. för att filtrera funktioner eller för att beräkna nya fältvärden. I vilket fall som helst måste uttrycket utvärderas. Det innebär att dess värde beräknas genom att utföra de angivna beräkningsstegen, som kan variera från enkel aritmetik till aggregerade uttryck.

11.2.1 Grundläggande uttryck

Detta grundläggande uttryck utvärderar en enkel aritmetisk operation:

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

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

Uttrycket kan också användas för jämförelse och utvärderas till 1 (`True`) eller 0 (`False`)

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

11.2.2 Uttryck med funktioner

För att utvärdera ett uttryck mot en funktion måste ett `QgsExpressionContext`-objekt skapas och skickas till evaluate-funktionen för att uttrycket ska få tillgång till funktionens fältvärden.

I följande exempel visas hur man skapar en funktion med ett fält som heter ”Column” och hur man lägger till denna funktion i uttryckskontexten.

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

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

Följande är ett mer komplett exempel på hur man använder uttryck i samband med ett vektorlager för att beräkna nya fältvärden:

```
from qgis.PyQt.QtCore import QMetaType

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

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

```

(continues on next page)

(fortsättning från föregående sida)

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

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

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

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

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

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

```
876.5
```

11.2.3 Filtrera ett lager med uttryck

Följande exempel kan användas för att filtrera ett lager och returnera alla funktioner som matchar ett predikat.

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

layer.startEditing()

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

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

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

(continues on next page)

(fortsättning från föregående sida)

```
    matches += 1

print(matches)
```

```
7
```

11.3 Hantering av uttrycksfel

Uttrycksrelaterade fel kan uppstå under parsning eller utvärdering av uttryck:

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

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

KAPITEL 12

Läsa och lagra inställningar

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

Många gånger är det användbart för ett plugin att spara vissa variabler så att användaren inte behöver ange eller välja dem igen nästa gång tillägget körs.

Dessa variabler kan sparas och hämtas med hjälp av Qt och QGIS API. För varje variabel bör du välja en nyckel som ska användas för att komma åt variabeln — för användarens favoritfärg kan du använda nyckeln ”favorite_color” eller någon annan meningsfull sträng. Vi rekommenderar att du ger namngivningen av nycklar en viss struktur.

Vi kan skilja mellan flera olika typer av inställningar:

- **globala inställningar** — de är knutna till användaren på en viss maskin. QGIS själv lagrar en hel del globala inställningar, till exempel huvudfönstrets storlek eller standardtolerans för snapping. Inställningar hanteras med hjälp av klassen `QgsSettings`, till exempel genom metoderna `setValue()` och `value()`.

  Här kan du se ett exempel på hur dessa metoder används.

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

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

(continues on next page)

(fortsättning från föregående sida)

```
  print(myint)
  print(myreal)
  print(nonexistent)
```

Den andra parametern i metoden `value()` är valfri och anger det standardvärde som returneras om det inte finns något tidigare värde angivet för det passerade inställningsnamnet.

För en metod att förkonfigurera standardvärdena för de globala inställningarna genom filen `qgis_global_settings.ini`, se deploying_organization för ytterligare information.

- **projektinställningar** — varierar mellan olika projekt och därför är de kopplade till en projektfil. Bakgrundsfärgen på kartduken eller destinationens koordinatreferenssystem (CRS) är exempel på detta — vit bakgrund och WGS84 kan vara lämpligt för ett projekt, medan gul bakgrund och UTM-projektion är bättre för ett annat.

  Ett exempel på användning följer nedan.

```
proj = QgsProject.instance()

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

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

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

Som du kan se används metoden `writeEntry()` för många datatyper (heltal, sträng, lista), men det finns flera metoder för att läsa tillbaka inställningsvärdet, och motsvarande måste väljas för varje datatyp.

- **inställningar för kartlager** — dessa inställningar är relaterade till en viss instans av ett kartlager med ett projekt. De är *inte* kopplade till den underliggande datakällan för ett lager, så om du skapar två kartlagerinstanser av en shapefil kommer de inte att dela inställningarna. Inställningarna lagras i projektfilen, så om användaren öppnar projektet igen kommer de lagerrelaterade inställningarna att finnas där igen. Värdet för en given inställning hämtas med metoden `customProperty()` och kan ställas in med metoden `setCustomProperty()`.

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

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

KAPITEL 13

Kommunicera med användaren

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

from qgis.gui import (
    QgsMessageBar,
)

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

I det här avsnittet beskrivs några metoder och element som bör användas för att kommunicera med användaren, så att användargränssnittet blir konsekvent.

13.1 Visning av meddelanden. Klassen QgsMessageBar

Att använda meddelanderutor kan vara en dålig idé ur användarupplevelsesynpunkt. För att visa en liten informationsrad eller ett varnings-/felmeddelande är QGIS meddelandefält vanligtvis ett bättre alternativ.

Med hjälp av referensen till QGIS-gränssnittsobjektet kan du visa ett meddelande i meddelandefältet med följande kod

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

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

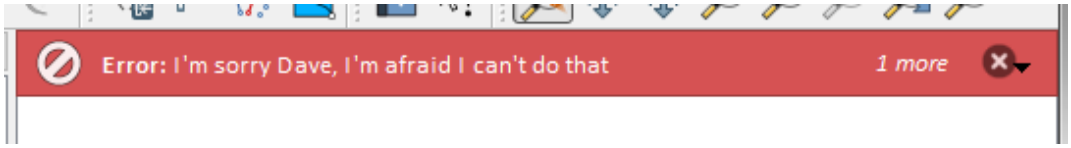


Fig. 13.1: QGIS meddelandefält

Du kan ställa in en varaktighet för att visa den under en begränsad tid

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

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

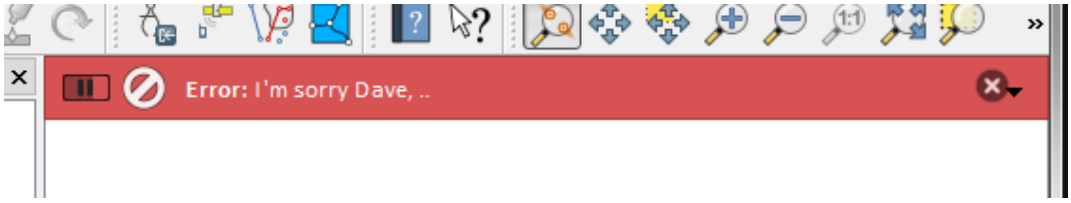


Fig. 13.2: QGIS meddelandefält med timer

Exemplen ovan visar en felrad, men parametern `level` kan användas för att skapa varningsmeddelanden eller informationsmeddelanden med hjälp av `Qgis.MessageLevel` uppräkning. Du kan använda upp till 4 olika nivåer:

0. Info
1. Varning
2. Kritiskt
3. Lyckades

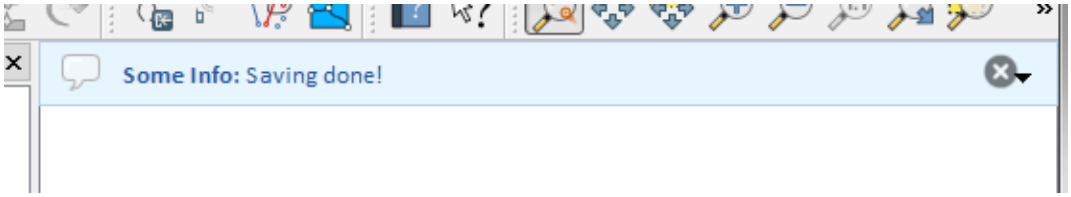


Fig. 13.3: QGIS meddelandefält (info)

Widgets kan läggas till i meddelandefältet, t.ex. en knapp för att visa mer information

```
def showError():
    pass

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

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

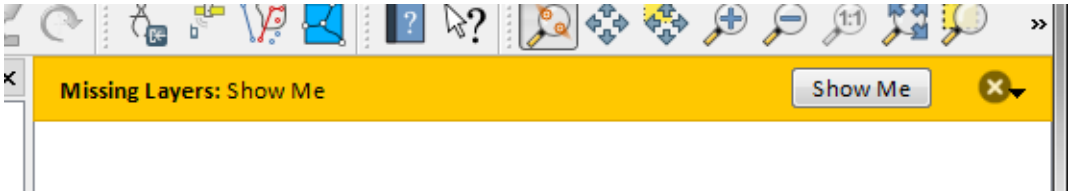


Fig. 13.4: QGIS meddelandefält med en knapp

Du kan även använda ett meddelandefält i din egen dialogruta så att du inte behöver visa en meddelanderuta, eller om det inte är meningsfullt att visa den i QGIS huvudfönster

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

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

Fig. 13.5: QGIS meddelandefält i anpassad dialog

13.2 Visar framsteg

Progressfält kan också placeras i QGIS meddelandefält, eftersom det, som vi har sett, accepterar widgetar. Här är ett exempel som du kan prova i konsolen.

```
import time
from qgis.PyQt.QtWidgets import QProgressBar
from qgis.PyQt.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)

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

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

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

Du kan också använda den inbyggda statusfältet för att rapportera framsteg, som i nästa exempel:

```
vlayer = iface.activeLayer()

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

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

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

iface.statusBarIface().clearMessage()
```

13.3 Loggning

Det finns tre olika typer av loggning tillgängliga i QGIS för att logga och spara all information om körningen av din kod. Var och en har sin specifika utmatningsplats. Tänk på att använda rätt loggningssätt för ditt ändamål:

- `QgsMessageLog` är för meddelanden för att kommunicera problem till användaren. Utdata från QgsMessageLog visas i panelen för loggmeddelanden.
- Den i Python inbyggda **logging**-modulen är avsedd för felsökning på samma nivå som QGIS Python API (PyQGIS). Den rekommenderas för Python-skriptutvecklare som behöver felsöka sin Python-kod, t.ex. objekt-ID:n eller geometrier
- `QgsLogger` är för meddelanden för *QGIS intern* felsökning / utvecklare (dvs. du misstänker att något utlöses av någon trasig kod). Meddelanden är endast synliga med utvecklarversioner av QGIS.

Exempel på de olika loggningstyperna visas i följande avsnitt nedan.

> **Varning**
>
> Användning av Pythons `print`-sats är osäkert att göra i all kod som kan vara flertrådad och **extremt saktar ner algoritmen**. Detta inkluderar **uttrycksfunktioner**, **renderare**, **symbollager** och **behandlingsalgoritmer** (bland andra). I dessa fall bör du alltid använda pythonmodulen **logging** eller trådsäkra klasser (`QgsLogger` eller `QgsMessageLog`) istället.

13.3.1 QgsMessageLog

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

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

> **Observera**
>
> Du kan se utdata från `QgsMessageLog` i log_message_panel

13.3.2 Den inbyggda loggningsmodulen i Python

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

Metoden basicConfig konfigurerar den grundläggande inställningen för loggningen. I koden ovan definieras filnamn, loggningsnivå och format. Filnamnet anger var loggfilen ska skrivas, loggningsnivån definierar vilka nivåer som ska matas ut och formatet definierar i vilket format varje meddelande ska matas ut.

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

Om du vill radera loggfilen varje gång du kör ditt skript kan du göra något liknande:

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

Mer information om hur du använder python-loggningsfunktionen finns på

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

> **Varning**
>
> Observera att om du inte loggar till en fil genom att ange ett filnamn kan loggningen vara flertrådad, vilket gör att utdata går långsammare.

KAPITEL 14

Infrastruktur för autentisering

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

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

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

from qgis.PyQt.QtNetwork import QSslCertificate
```

14.1 Introduktion

Användarreferenser för autentiseringsinfrastrukturen kan läsas i användarhandboken i avsnittet authentication_overview.

I det här kapitlet beskrivs de bästa metoderna för att använda Authentication-systemet ur ett utvecklarperspektiv.

Autentiseringssystemet används ofta i QGIS Desktop av dataleverantörer när det krävs autentiseringsuppgifter för att komma åt en viss resurs, t.ex. när ett lager upprättar en anslutning till en Postgres-databas.

Det finns också några widgetar i QGIS gui-bibliotek som tilläggsutvecklare kan använda för att enkelt integrera autentiseringsinfrastrukturen i sin kod:

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

En bra kodreferens kan läsas från autentiseringsinfrastrukturen tests code.

> **Varning**
>
> På grund av de säkerhetsbegränsningar som beaktades under utformningen av autentiseringsinfrastrukturen exponeras endast en utvald delmängd av de interna metoderna för Python.

14.2 Ordlista

Här följer några definitioner av de vanligaste objekten som behandlas i detta kapitel.

Huvudlösenord
: Lösenord för att tillåta åtkomst och dekryptera autentiseringsuppgifter som lagras i QGIS Authentication DB

Autentiseringsdatabas
: En *Master Password* krypterad sqlite db `qgis-auth.db` där *Authentication Configuration* lagras. t.ex. användare/lösenord, personliga certifikat och nycklar, certifikatutfärdare

Autentisering DB
: *Authentication Database*

Konfiguration av autentisering
: En uppsättning autentiseringsdata beroende på *Authentication Method*. t.ex. Basic authentication method lagrar paret användare/lösenord.

Konfig för autentisering
: *Authentication Configuration*

Autentiseringsmetod
: En specifik metod som används för att bli autentiserad. Varje metod har sitt eget protokoll som används för att uppnå den autentiserade nivån. Varje metod är implementerad som ett delat bibliotek som laddas dynamiskt under initieringen av QGIS autentiseringsinfrastruktur.

14.3 QgsAuthManager ingångspunkten

Singletonen `QgsAuthManager` är ingångspunkten för att använda de autentiseringsuppgifter som lagras i QGIS krypterade *Authentication DB*, dvs. filen `qgis-auth.db` under den aktiva mappen user profile.

Denna klass tar hand om användarinteraktionen: genom att be om att få ange ett huvudlösenord eller genom att på ett transparent sätt använda det för att komma åt krypterad lagrad information.

14.3.1 Starta chefen och ställ in huvudlösenordet

Följande kodavsnitt ger ett exempel på hur du anger huvudlösenord för att öppna åtkomsten till autentiseringsinställningarna. Kodkommentarerna är viktiga för att förstå utdraget.

```
authMgr = QgsApplication.authManager()

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

14.3.2 Fyll authdb med en ny post för konfiguration av autentisering

Alla lagrade referenser är en *Authentication Configuration*-instans av `QgsAuthMethodConfig`-klassen som nås med hjälp av en unik sträng som den följande:

```
authcfg = 'fm1s770'
```

denna sträng genereras automatiskt när en post skapas med QGIS API eller GUI, men det kan vara bra att manuellt ställa in den på ett känt värde om konfigurationen måste delas (med olika autentiseringsuppgifter) mellan flera användare inom en organisation.

`QgsAuthMethodConfig` är basklassen för alla *Authentication Method*. Varje autentiseringsmetod anger en konfigurationshashkarta där autentiseringsinformation lagras. Nedan följer ett användbart utdrag för att lagra PKI-path-autentiseringsuppgifter för en hypotetisk alice-användare:

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

(continues on next page)

(fortsättning från föregående sida)

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

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

Tillgängliga autentiseringsmetoder

Authentication Method-bibliotek laddas dynamiskt under init av autentiseringshanteraren. Tillgängliga autentiseringsmetoder är:

1. `Basic` Autentisering av användare och lösenord
2. `EsriToken` ESRI tokenbaserad autentisering
3. `Identity-Cert` Autentisering av identitetscertifikat
4. `OAuth2` OAuth2-autentisering
5. `PKI-Paths` PKI-sökvägar autentisering
6. `PKI-PKCS#12` PKI PKCS#12 autentisering

Fylla i myndigheter

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

Hantera PKI-bundlar med QgsPkiBundle

En praktisk klass för att packa PKI-bundlar som består av SslCert, SslKey och CA-kedja är `QgsPkiBundle`-klassen. Nedan följer ett utdrag för att få lösenordsskydd:

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

Se `QgsPkiBundle` klassdokumentation för att extrahera cert/key/CAs från paketet.

14.3.3 Ta bort en post från authdb

Vi kan ta bort en post från *Authentication Database* med hjälp av dess `authcfg` identifierare med följande utdrag:

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

14.3.4 Lämna authcfg-utvidgningen till QgsAuthManager

Det bästa sättet att använda en *Authentication Config* som finns lagrad i *Authentication DB* är att referera till den med den unika identifieraren `authcfg`. Expandera innebär att konvertera den från en identifierare till en komplett uppsättning autentiseringsuppgifter. Den bästa metoden för att använda lagrade *Authentication Config* är att låta den hanteras automatiskt av Authentication Manager. Den vanligaste användningen av en lagrad konfiguration är att ansluta till en autentiseringsaktiverad tjänst som en WMS eller WFS eller till en DB-anslutning.

Observera

Tänk på att inte alla QGIS-dataleverantörer är integrerade med autentiseringsinfrastrukturen. Varje autentiseringsmetod härstammar från basklassen `QgsAuthMethod` och stöder en annan uppsättning leverantörer. Till exempel stöder metoden `certIdentity()` följande lista med leverantörer:

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

Exempel på utdata:

```
['ows', 'wfs', 'wcs', 'wms', 'postgres']
```

Om du t.ex. vill komma åt en WMS-tjänst med hjälp av lagrade autentiseringsuppgifter som identifierats med `authcfg = 'fm1s770'`, behöver du bara använda `authcfg` i datakällans URL som i följande utdrag:

```
authCfg = 'fm1s770'
quri = QgsDataSourceUri()
quri.setParam("layers", 'usa:states')
quri.setParam("styles", '')
quri.setParam("format", 'image/png')
quri.setParam("crs", 'EPSG:4326')
quri.setParam("dpiMode", '7')
quri.setParam("featureCount", '10')
quri.setParam("authcfg", authCfg)    # <---- here my authCfg url parameter
quri.setParam("contextualWMSLegend", '0')
quri.setParam("url", 'https://my_auth_enabled_server_ip/wms')
rlayer = QgsRasterLayer(str(quri.encodedUri(), "utf-8"), 'states', 'wms')
```

I det övre fallet kommer leverantören `wms` att se till att expandera URI-parametern `authcfg` med referenser precis innan HTTP-anslutningen ställs in.

Varning

Utvecklaren skulle behöva lämna `authcfg` expansion till `QgsAuthManager`, på detta sätt kommer han att vara säker på att expansion inte görs för tidigt.

Vanligtvis används en URI-sträng, byggd med hjälp av klassen `QgsDataSourceURI`, för att ange en datakälla på följande sätt:

```
authCfg = 'fm1s770'
quri = QgsDataSourceUri("my WMS uri here")
```

(continues on next page)

(fortsättning från föregående sida)

```
quri.setParam("authcfg", authCfg)
rlayer = QgsRasterLayer( quri.uri(False), 'states', 'wms')
```

Observera

Parametern `False` är viktig för att undvika fullständig URI-expansion av id:t `authcfg` som finns i URI:n.

PKI-exempel med andra dataleverantörer

Andra exempel kan läsas direkt i QGIS-testerna uppströms som i test_authmanager_pki_ows eller test_authmanager_pki_postgres.

14.4 Anpassa tillägg för att använda autentiseringsinfrastruktur

Många tillägg från tredje part använder httplib2 eller andra Python-nätverksbibliotek för att hantera HTTP-anslutningar istället för att integrera med `QgsNetworkAccessManager` och dess relaterade Authentication Infrastructure-integration.

För att underlätta denna integration har en Python-hjälpfunktion skapats som heter `NetworkAccessManager`. Dess kod kan hittas här.

Denna hjälpklass kan användas som i följande snippet:

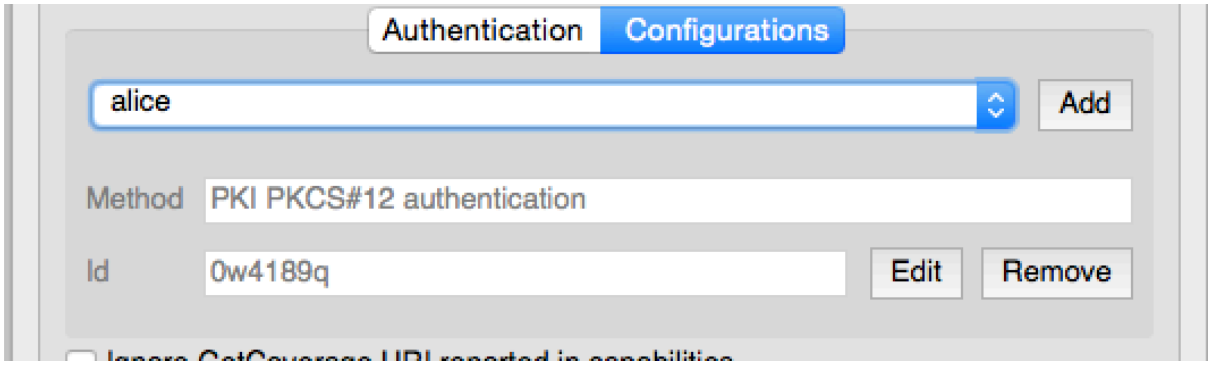
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 Grafiska gränssnitt för autentisering

I detta stycke listas de tillgängliga grafiska användargränssnitten som är användbara för att integrera autentiseringsinfrastruktur i anpassade gränssnitt.

14.5.1 Grafiskt gränssnitt för att välja autentiseringsuppgifter

Om det är nödvändigt att välja en *Authentication Configuration* från den uppsättning som lagras i *Authentication DB* finns den tillgänglig i GUI-klassen `QgsAuthConfigSelect`.



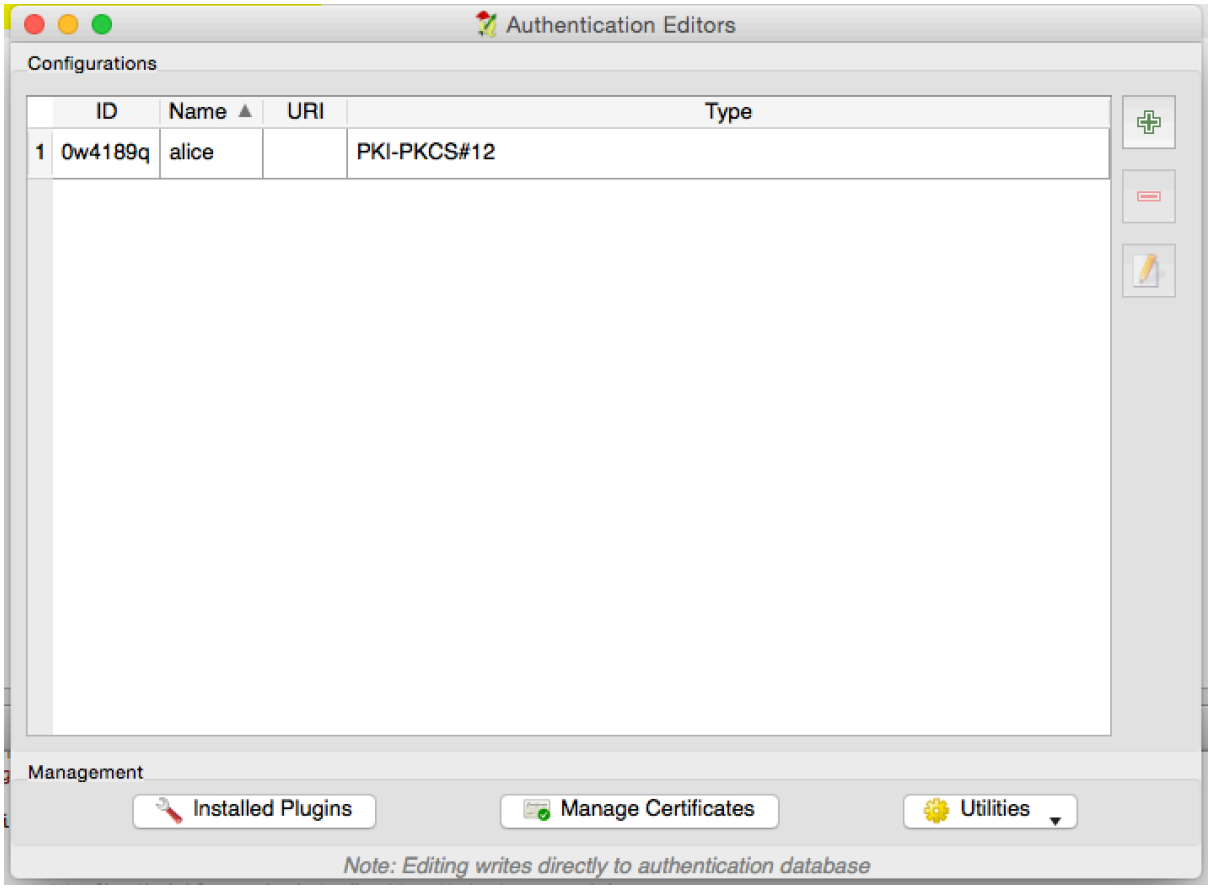
och kan användas på samma sätt som i följande utdrag:

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

Exemplet ovan är hämtat från QGIS-källan code. Den andra parametern i GUI-konstruktören hänvisar till typen av dataleverantör. Parametern används för att begränsa de kompatibla *Authentication Method*'s med den angivna leverantören.

14.5.2 Grafiskt gränssnitt för autentiseringsredigerare

Hela det grafiska gränssnittet som används för att hantera autentiseringsuppgifter, auktoriteter och för åtkomst till autentiseringsverktyg hanteras av klassen `QgsAuthEditorWidgets`.



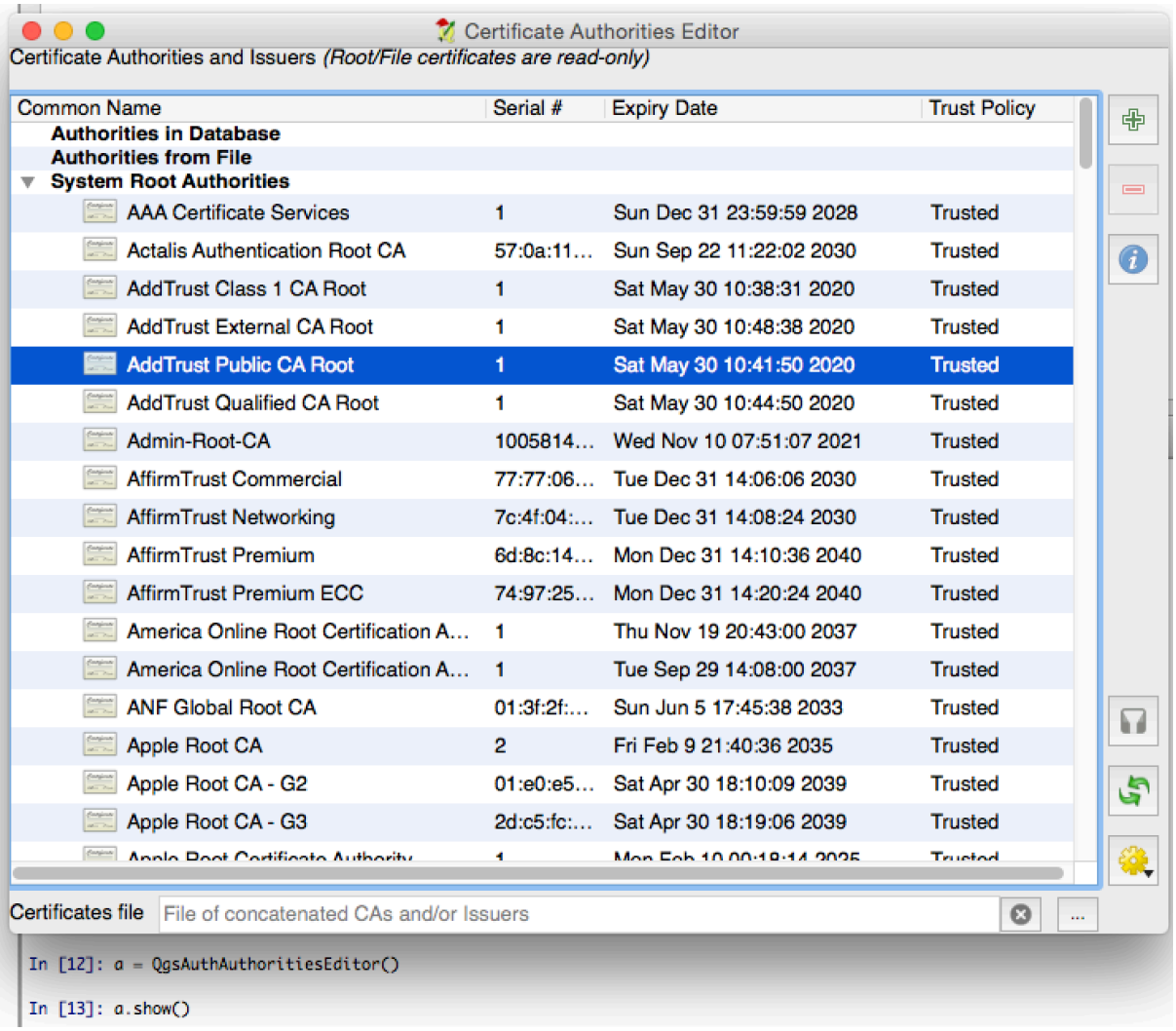
och kan användas på samma sätt som i följande utdrag:

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

Ett integrerat exempel finns i den relaterade test.

14.5.3 Grafiskt gränssnitt för behörighetsredigerare

Ett grafiskt gränssnitt som används för att hantera endast behörigheter hanteras av klassen `QgsAuthAuthoritiesEditor`.



och kan användas på samma sätt som i följande utdrag:

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

KAPITEL 15

Tasks - utför tungt arbete i bakgrunden

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

15.1 Introduktion

Bakgrundsbearbetning med hjälp av trådar är ett sätt att bibehålla ett responsivt användargränssnitt när tung bearbetning pågår. Tasks kan användas för att uppnå trådning i QGIS.

En uppgift (`QgsTask`) är en behållare för den kod som ska utföras i bakgrunden, och uppgiftshanteraren (`QgsTaskManager`) används för att styra körningen av uppgifterna. Dessa klasser förenklar bakgrundsbearbetningen i QGIS genom att tillhandahålla mekanismer för signalering, förloppsrapportering och åtkomst till status för bakgrundsprocesser. Uppgifter kan grupperas med hjälp av underuppgifter.

Den globala uppgiftshanteraren (hittas med `QgsApplication.taskManager()`) används normalt. Detta innebär att dina uppgifter kanske inte är de enda uppgifter som styrs av uppgiftshanteraren.

Det finns flera sätt att skapa en QGIS-uppgift:

- Skapa din egen uppgift genom att utöka `QgsTask`

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

- Skapa en uppgift från en funktion

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

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

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

- Skapa en uppgift från en bearbetningsalgoritm

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

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

Varning

En bakgrundsuppgift (oavsett hur den skapas) får ALDRIG använda något QObject som finns i huvudtråden, t.ex. QgsVectorLayer, QgsProject eller utföra grafiska gränssnittsbaserade operationer som att skapa nya widgetar eller interagera med befintliga widgetar. Qt-widgetar får endast nås eller ändras från huvudtråden. Data som används i en uppgift måste kopieras innan uppgiften startas. Försök att använda dem från bakgrundstrådar kommer att leda till krascher.

Se dessutom alltid till att `context` och `feedback` lever minst lika länge som de uppgifter som använder dem. QGIS kommer att krascha om *QgsTaskManager*, när en uppgift har slutförts, misslyckas med att komma åt *kontexten* och *feedback* som uppgiften schemalades mot.

Observera

Det är ett vanligt mönster att anropa `setProject()` strax efter anropet av `QgsProcessingContext`. Detta gör att uppgiften och dess återuppringningsfunktion kan använda de flesta av de projektomfattande inställningarna. Detta är särskilt värdefullt när man arbetar med spatiala lager i callback-funktionen.

Beroenden mellan uppgifter kan beskrivas med hjälp av funktionen `addSubTask()` i `QgsTask`. När ett beroende anges kommer uppgiftshanteraren automatiskt att avgöra hur dessa beroenden ska utföras. När det är möjligt kommer beroenden att utföras parallellt för att uppfylla dem så snabbt som möjligt. Om en uppgift som en annan uppgift är beroende av avbryts, kommer även den beroende uppgiften att avbrytas. Cirkulära beroenden kan leda till deadlocks, så var försiktig.

Om en uppgift är beroende av att ett lager är tillgängligt kan detta anges med funktionen `setDependentLayers()` i `QgsTask`. Om ett lager som en uppgift är beroende av inte är tillgängligt kommer uppgiften att avbrytas.

När uppgiften har skapats kan den schemaläggas för körning med hjälp av funktionen `addTask()` i uppgiftshanteraren. Genom att lägga till en uppgift i hanteraren överförs äganderätten till uppgiften automatiskt till hanteraren, och hanteraren rensar och tar bort uppgifter efter att de har körts. Schemaläggningen av uppgifterna påverkas av uppgiftsprioriteten, som ställs in i `addTask()`.

Status för uppgifter kan övervakas med hjälp av signaler och funktioner i `QgsTask` och `QgsTaskManager`.

15.2 Exempel

15.2.1 Utökning av QgsTask

I det här exemplet utökar `RandomIntegerSumTask` `QgsTask` och kommer att generera 100 slumpmässiga heltal mellan 0 och 500 under en angiven tidsperiod. Om det slumpmässiga talet är 42 avbryts uppgiften och ett undantag tas upp. Flera instanser av `RandomIntegerSumTask` (med underuppgifter) genereras och läggs till i uppgiftshanteraren, vilket demonstrerar två typer av beroenden.

```
import random
from time import sleep

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

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""

    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None

    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
        Raising exceptions will crash QGIS, so we handle them
        internally and raise them in self.finished
        """
        QgsMessageLog.logMessage('Started task "{}"'.format(
                                    self.description()),
                                 MESSAGE_CATEGORY, Qgis.Info)
        wait_time = self.duration / 100
        for i in range(100):
            sleep(wait_time)
            # use setProgress to report progress
            self.setProgress(i)
            arandominteger = random.randint(0, 500)
            self.total += arandominteger
            self.iterations += 1
            # check isCanceled() to handle cancellation
            if self.isCanceled():
                return False
            # simulate exceptions to show how to abort task
            if arandominteger == 42:
                # DO NOT raise Exception('bad value!')
                # this would crash QGIS
                self.exception = Exception('bad value!')
                return False
        return True

    def finished(self, result):
        """
        This function is automatically called when the task has
```

(continues on next page)

(fortsättning från föregående sida)

```
        completed (successfully or not).
        You implement finished() to do whatever follow-up stuff
        should happen after the task is complete.
        finished is always called from the main thread, so it's safe
        to do GUI operations and raise Python exceptions here.
        result is the return value from self.run.
        """
        if result:
            QgsMessageLog.logMessage(
                'RandomTask "{name}" completed\n' \
                'RandomTotal: {total} (with {iterations} '\
              'iterations)'.format(
                  name=self.description(),
                  total=self.total,
                  iterations=self.iterations),
                MESSAGE_CATEGORY, Qgis.Success)
        else:
            if self.exception is None:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" not successful but without '\
                    'exception (probably the task was manually '\
                    'canceled by the user)'.format(
                        name=self.description()),
                    MESSAGE_CATEGORY, Qgis.Warning)
            else:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" Exception: {exception}'.format(
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception

    def cancel(self):
        QgsMessageLog.logMessage(
            'RandomTask "{name}" was canceled'.format(
                name=self.description()),
            MESSAGE_CATEGORY, Qgis.Info)
        super().cancel()


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

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

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

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

15.2.2 Uppgift från funktion

Skapa en uppgift från en funktion (`doSomething` i det här exemplet). Funktionens första parameter kommer att innehålla `QgsTask` för funktionen. En viktig (namngiven) parameter är `on_finished`, som anger en funktion som ska anropas när uppgiften har slutförts. Funktionen `doSomething` i det här exemplet har ytterligare en namngiven parameter `wait_time`.

```
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

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

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

(continues on next page)

(fortsättning från föregående sida)

```
        MESSAGE_CATEGORY, Qgis.Info)

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

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

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

15.2.3 Uppgift från en bearbetningsalgoritm

Skapa en uppgift som använder algoritmen qgis:randompointsinextent för att generera 50000 slumpmässiga punkter inom en angiven utsträckning. Resultatet läggs till i projektet på ett säkert sätt.

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

MESSAGE_CATEGORY = 'AlgRunnerTask'

def task_finished(context, successful, results):
    if not successful:
        QgsMessageLog.logMessage('Task finished unsucessfully',
                                 MESSAGE_CATEGORY, Qgis.Warning)
```

(continues on next page)

(fortsättning från föregående sida)

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

alg = QgsApplication.processingRegistry().algorithmById(
                                       'qgis:randompointsinextent')
# `context` and `feedback` need to
# live for as least as long as `task`,
# otherwise the program will crash.
# Initializing them globally is a sure way
# of avoiding this unfortunate situation.
context = QgsProcessingContext()
feedback = QgsProcessingFeedback()
params = {
    'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
    'MIN_DISTANCE': 0.0,
    'POINTS_NUMBER': 50000,
    'TARGET_CRS': 'EPSG:4326',
    'OUTPUT': 'memory:My random points'
}
task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
task.executed.connect(partial(task_finished, context))
QgsApplication.taskManager().addTask(task)
```

Se även: https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/.

KAPITEL 16

Utveckla Python-tillägg

16.1 Strukturering av Python-tillägg

De viktigaste stegen för att skapa ett plugin är:

1. *Idé*: Ha en idé om vad du vill göra med ditt nya QGIS-plugin.
2. *Setup*: *Skapa filerna för ditt tillägg*. Beroende på typen av plugin är vissa obligatoriska medan andra är valfria
3. *Develop*: *Skriv koden* i lämpliga filer
4. *Dokument*: *Skriva dokumentationen för tillägget*
5. Valfritt: *Translate*: *Översätt ditt tillägg* till olika språk
6. *Test*: *Ladda om ditt tillägg* för att kontrollera om allt är OK
7. *Publicera*: Publicera din plugin i QGIS repository eller skapa din egen repository som en ”arsenal” av personliga ”GIS-vapen”.

16.1.1 Kom igång

Innan du börjar skriva ett nytt tillägg, ta en titt på *Officiellt Python-tilläggsarkiv*. Källkoden för befintliga tillägg kan hjälpa dig att lära dig mer om programmering. Du kanske också upptäcker att ett liknande plugin redan finns och att du kanske kan utöka det eller åtminstone bygga vidare på det för att utveckla ditt eget.

Konfigurera plugin-filens struktur

För att komma igång med ett nytt plugin måste vi konfigurera de nödvändiga plugin-filerna.

Det finns två resurser för plugin-mallar som kan hjälpa dig att komma igång:

- För utbildningsändamål eller när ett minimalistiskt tillvägagångssätt önskas, tillhandahåller minimal plugin-mall de grundläggande filer (skelett) som krävs för att skapa ett giltigt QGIS Python-plugin.
- För en plugin-mall med mer fullständiga funktioner kan Plugin Builder skapa mallar för flera olika plugin-typer, inklusive funktioner som lokalisering (översättning) och testning.

En typisk plugin-katalog innehåller följande filer:

- `metadata.txt` - *krävs* - Innehåller allmän information, version, namn och några andra metadata som används av tillägg webbplats och plugin-infrastruktur.
- `__init__.py` - *krävs* - Startpunkten för tillägget. Den måste ha metoden `classFactory()` och kan ha annan initialiseringskod.
- `mainPlugin.py` - *kärnkod* - Den huvudsakliga arbetskoden för tillägget. Innehåller all information om pluginets åtgärder och huvudkoden.
- `form.ui` - *för tillägg med eget grafiskt gränssnitt* - Det grafiska gränssnitt som skapats av Qt Designer.
- `form.py` - *kompilerat grafiskt gränssnitt* - Översättningen av form.ui som beskrivs ovan till Python.
- `resources.qrc` - *valfritt* - Ett .xml-dokument som skapats av Qt Designer. Innehåller relativa sökvägar till resurser som används i de grafiska gränssnittsformulären.
- `resources.py` - *kompilerade resurser, valfritt* - Översättningen av den ovan beskrivna .qrc-filen till Python.
- `LICENSE` - *krävs* om tillägget ska publiceras eller uppdateras i QGIS Tillägg Directory, annars *valfritt*. Filen ska vara en ren textfil utan filtillägg i filnamnet.

> **Varning**
>
> Om du planerar att ladda upp tillägget till *Officiellt Python-tilläggsarkiv* måste du kontrollera att ditt tillägg följer några ytterligare regler, som krävs för tillägget *Validering*.

16.1.2 Skriva plugin-kod

I följande avsnitt visas vilket innehåll som ska läggas till i var och en av de filer som introducerades ovan.

metadata.txt

Först måste Tilläggshanterare hämta grundläggande information om tillägget, t.ex. dess namn, beskrivning osv. Denna information lagras i `metadata.txt`.

> **Observera**
>
> Alla metadata måste vara i UTF-8-kodning.

Namn på metadata	Obligato	Anteckningar
namn	Sant	en kort sträng som innehåller namnet på tillägget
qgisMinimumV	Sant	prickad notation av minsta QGIS-version
qgisMaximumV	Falskt	streckad notation av maximal QGIS-version
description	Sant	kort text som beskriver tillägget, ingen HTML tillåten
about	Sant	längre text som beskriver tillägget i detalj, ingen HTML tillåten
version	Sant	kort sträng med versionen prickad notation
author	Sant	författarens namn
email	Sant	författarens e-postadress, visas endast på webbplatsen för inloggade användare, men syns i Tilläggshanterare efter att tillägget har installerats
changelog	Falskt	sträng, kan vara flerradig, ingen HTML tillåten
experimental	Falskt	boolean flagga, `True` eller `False` - `True` om denna version är experimentell
deprecated	Falskt	boolesk flagga, `True` eller `False`, gäller för hela tillägget och inte bara för den uppladdade versionen
tags	Falskt	kommaseparerad lista, mellanslag är tillåtna inom enskilda taggar
homepage	Falskt	en giltig URL som pekar på hemsidan för ditt plugin
repository	Sant	en giltig URL för källkodsarkivet
tracker	Falskt	en giltig URL för ärenden och felrapporter
icon	Falskt	ett filnamn eller en relativ sökväg (i förhållande till basmappen i tilläggets komprimerade paket) för en webbvänlig bild (PNG, JPEG)
category	Falskt	en av `Raster`, `Vector`, `Database`, `Mesh` och `Web`
plugin_depend	Falskt	PIP-liknande kommaseparerad lista över andra tillägg som ska installeras, använd tilläggens namn från deras metadatas namnfält
server	Falskt	boolean flagga, `True` eller `False`, avgör om tillägget har ett servergränssnitt
hasProcessingP	Falskt	boolean flagga, `True` eller `False`, avgör om tillägget tillhandahåller bearbetningsalgoritmer

Som standard placeras tillägg i menyn *Tillägg* (vi kommer i nästa avsnitt att se hur du lägger till en menypost för ditt plugin) men de kan också placeras i menyerna *Raster*, *Vector*, *Database*, *Mesh* och *Web*.

En motsvarande ”kategori” metadatapost finns för att ange detta, så att tillägget kan klassificeras i enlighet därmed. Denna metadatapost används som tips för användare och talar om för dem var (i vilken meny) tillägget kan hittas. Tillåtna värden för ”category” är: Vektor, Raster, Databas eller Webb. Till exempel, om ditt plugin kommer att finnas tillgängligt från *Raster*-menyn, lägg till detta i `metadata.txt`

```
category=Raster
```

Observera

Om *qgisMaximumVersion* är tom, kommer den automatiskt att sättas till huvudversionen plus *.99* när den laddas upp till *Officiellt Python-tilläggsarkiv*.

Ett exempel på detta är metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=3.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
```

(continues on next page)

(fortsättning från föregående sida)

```
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
; end of mandatory metadata

; start of optional metadata
category=Raster
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

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

; these metadata can be empty, they will eventually become mandatory.
homepage=https://www.itopen.it
icon=icon.png

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

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

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=3.99

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

__init__.py

Denna fil krävs av Pythons importsystem. QGIS kräver också att denna fil innehåller en `classFactory()`-funktion, som anropas när tillägget laddas in i QGIS. Den får en referens till instansen av `QgisInterface` och måste returnera ett objekt av din tillägg klass från `mainplugin.py` — i vårt fall heter den `TestPlugin` (se nedan). Så här bör `__init__.py` se ut

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

# any other initialisation needed
```

mainPlugin.py

Det är här det magiska händer och det är så här det magiska ser ut: (t.ex. `mainPlugin.py`)

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

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

class TestPlugin:

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

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

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

    # connect to signal renderComplete which is emitted when canvas
    # rendering is done
    self.iface.mapCanvas().renderComplete.connect(self.renderTest)

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

    # disconnect form signal of the canvas
    self.iface.mapCanvas().renderComplete.disconnect(self.renderTest)

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

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

De enda plugin-funktioner som måste finnas i plugin-huvudkällfilen (t.ex. `mainPlugin.py`) är

- `__init__` som ger tillgång till QGIS-gränssnittet
- `initGui()` anropas när tillägget laddas
- `unload()` anropas när tillägget avlastas

I exemplet ovan används `addPluginToMenu()`. Detta kommer att lägga till motsvarande menyåtgärd till menyn *Tillägg*. Det finns alternativa metoder för att lägga till åtgärden i en annan meny. Här är en lista över dessa metoder:

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

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

Alla dessa har samma syntax som metoden `addPluginToMenu()`.

Att lägga till din plugin-meny till en av dessa fördefinierade metoder rekommenderas för att hålla konsekvens i hur plugin-poster organiseras. Du kan dock lägga till din anpassade menygrupp direkt i menyfältet, vilket nästa exempel visar:

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

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

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

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

Glöm inte att ställa in `QAction` och `QMenu` `objectName` till ett namn som är specifikt för ditt plugin så att det kan anpassas.

Även om hjälp och om-åtgärder också kan läggas till i din anpassade meny, är en bekväm plats att göra dem tillgängliga i QGIS huvudmeny :menuselection:`Help --> Tillägg`. Detta görs med hjälp av `pluginHelpMenu()`-metoden.

```
def initGui(self):

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

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

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

def unload(self):

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

När du arbetar med ett riktigt tillägg är det klokt att skriva tillägget i en annan (arbets)katalog och skapa en makefile som genererar UI + resursfiler och installerar tillägget i din QGIS-installation.

16.1.3 Dokumentera tillägg

Dokumentationen för tillägget kan skrivas som HTML-hjälpfiler. Modulen `qgis.utils` tillhandahåller en funktion, `showPluginHelp()`, som öppnar webbläsaren för hjälpfiler på samma sätt som annan QGIS-hjälp.

Funktionen `showPluginHelp()` letar efter hjälpfiler i samma katalog som den anropande modulen. Den letar i tur och ordning efter `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` och `index.html`, och visar den som hittas först. Här är `ll_cc` QGIS-landets språk. Detta gör att flera översättningar av dokumentationen kan inkluderas med tillägget.

Funktionen `showPluginHelp()` kan också ta parametrar packageName, som identifierar ett specifikt plugin för vilket hjälpen ska visas, filename, som kan ersätta ”index” i namnen på de filer som söks, och section, som är namnet på en html-ankartagg i det dokument som webbläsaren ska placeras på.

16.1.4 Översättning av tillägg

Med några få steg kan du ställa in miljön för plugin-lokalisering så att tillägget laddas på olika språk beroende på lokalinställningarna på din dator.

Krav på programvara

Det enklaste sättet att skapa och hantera alla översättningsfiler är att installera Qt Linguist. I en Debian-baserad GNU/Linux-miljö kan du installera den genom att skriva:

```
sudo apt install qttools5-dev-tools
```

Filer och kataloger

När du skapar tillägget hittar du mappen `i18n` i huvudkatalogen för tillägget.

Alla översättningsfiler måste finnas i den här katalogen.

.pro-fil

Först bör du skapa en `.pro`-fil, det vill säga en *projekt*-fil som kan hanteras av **Qt Linguist**.

I den här filen `.pro` måste du ange alla filer och formulär som du vill översätta. Den här filen används för att konfigurera lokaliseringsfilerna och variablerna. En möjlig projektfil som matchar strukturen i vårt *exempelplugin*:

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

Ditt tillägg kan ha en mer komplex struktur och kan vara distribuerat över flera filer. Om så är fallet, kom ihåg att `pylupdate5`, det program vi använder för att läsa filen `.pro` och uppdatera den översättningsbara strängen, inte expanderar jokertecken, så du måste placera varje fil uttryckligen i filen `.pro`. Din projektfil kan då se ut ungefär så här:

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

Dessutom är filen `your_plugin.py` den fil som *kallar* alla menyer och undermenyer för ditt plugin i QGIS verktygsfält och du vill översätta dem alla.

Slutligen kan du med variabeln *TRANSLATIONS* ange vilka översättningsspråk du vill ha.

> **Varning**
>
> Var noga med att namnge filen `ts` som `ditt_plugin_` + `language` + `.ts` annars kommer språkinläsningen att misslyckas! Använd genvägen med 2 bokstäver för språket (**it** för italienska, **de** för tyska, etc …)

.ts fil

När du har skapat `.pro` är du redo att generera `.ts`-filen för språken i ditt tillägg.

Öppna en terminal, gå till katalogen `your_plugin/i18n` och skriv:

```
pylupdate5 your_plugin.pro
```

bör du se filen `your_plugin_language.ts`.

Öppna filen `.ts` med **Qt Linguist** och börja översätta.

.qm-fil

När du är klar med att översätta ditt tillägg (om vissa strängar inte är färdiga kommer källspråket för dessa strängar att användas) måste du skapa filen `.qm` (den kompilerade `.ts`-filen som kommer att användas av QGIS).

Öppna bara en terminal cd i katalogen `your_plugin/i18n` och skriv:

```
lrelease your_plugin.ts
```

nu, i katalogen `i18n` kommer du att se filen `ditt_tillägg.qm`.

Översätt med hjälp av Makefile

Alternativt kan du använda makefilen för att extrahera meddelanden från pythonkod och Qt-dialogrutor, om du har skapat ditt tillägg med Plugin Builder. I början av Makefile finns det en LOCALES-variabel:

```
LOCALES = en
```

Lägg till förkortningen för språket i denna variabel, t.ex. för ungerska:

```
LOCALES = en hu
```

Nu kan du generera eller uppdatera filen `hu.ts` (och även `en.ts`) från källorna genom att:

```
make transup
```

Efter detta har du uppdaterat filen `.ts` för alla språk som anges i variabeln LOCALES. Använd **Qt Linguist** för att översätta programmeddelandena. När översättningen är klar kan filerna `.qm` skapas av transcompile:

```
make transcompile
```

Du måste distribuera `.ts`-filer med ditt tillägg.

Ladda tillägget

För att se översättningen av ditt tillägg, öppna QGIS, ändra språk (*Inställningar* ► *Alternativ* ► *Allmänt*) och starta om QGIS.

Du bör se ditt tillägg på rätt språk.

> **Varning**
>
> Om du ändrar något i ditt tillägg (nya UIs, ny meny, etc..) måste du ** generera igen ** uppdateringsversionen av både `.ts` och `.qm`-filen, så kör kommandot ovan igen.

16.1.5 Dela ditt tillägg

QGIS är värd för hundratals tillägg i tilläggsarkivet. Överväg att dela med dig av din! Det kommer att utöka möjligheterna med QGIS och människor kommer att kunna lära sig av din kod. Alla tillägg som finns i QGIS kan hittas och installeras från QGIS med tilläggshanteraren.

Information och krav finns här: tillägg.qgis.org.

16.1.6 Tips och tricks

Omladdare för tillägg

Under utvecklingen av ditt tillägg kommer du ofta att behöva ladda om det i QGIS för testning. Detta är mycket enkelt med hjälp av tillägget **Plugin Reloader**. Du kan hitta det med Tilläggshanterare.

Automatisera paketering, release och översättning med qgis-plugin-ci

qgis-plugin-ci tillhandahåller ett kommandoradsgränssnitt för att utföra automatiserad paketering och distribution för QGIS-tillägg på din dator, eller med hjälp av kontinuerlig integration som GitHub workflows eller Gitlab-CI samt Transifex för översättning.

Det gör det möjligt att släppa, översätta, publicera eller generera en XML-tilläggsarkivfil via CLI eller i CI-åtgärder.

Tillgång till tillägg

Du kan komma åt alla klasser av installerade tillägg från QGIS med hjälp av python, vilket kan vara praktiskt för felsökning.

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

Loggmeddelanden

Tillägg har en egen flik inom log_message_panel.

Resursfil

Vissa tillägg använder resursfiler, t.ex. `resources.qrc` som definierar resurser för grafiska gränssnitt, t.ex. ikoner:

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

Det är bra att använda ett prefix som inte kolliderar med andra tillägg eller delar av QGIS, annars kan du få resurser som du inte vill ha. Nu behöver du bara generera en Python-fil som kommer att innehålla resurserna. Det görs med kommandot **pyrcc5**:

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

Observera

I Windows-miljöer kommer försök att köra **pyrcc5** från Command Prompt eller Powershell förmodligen att resultera i felet ”Windows kan inte komma åt den angivna enheten, sökvägen eller filen [...]”. Den enklaste lösningen är förmodligen att använda OSGeo4W Shell men om du känner dig bekväm med att ändra PATH-miljövariabeln eller ange sökvägen till den körbara filen uttryckligen bör du kunna hitta den på `<Your QGIS Install Directory>\bin\pyrcc5.exe`.

16.2 Kodsnuttar

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

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

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

Detta avsnitt innehåller kodavsnitt som underlättar plugin-utveckling.

16.2.1 Hur man anropar en metod med en kortkommando

I tillägget lägger du till `initGui()`

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

Till `unload()` lägg till

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

Den metod som anropas när CTRL+I trycks in

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

Det är också möjligt att låta användare anpassa kortkommandon för de åtgärder som tillhandahålls. Detta görs genom att lägga till:

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

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

16.2.2 Hur man återanvänder QGIS-ikoner

Eftersom de är välkända och förmedlar ett tydligt budskap till användarna kanske du ibland vill återanvända QGIS-ikoner i ditt plugin istället för att rita och ställa in en ny. Använd metoden `getThemeIcon()`.

Till exempel för att återanvända ikonen  mActionFileOpen.svg som finns i QGIS kodarkiv:

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

`iconPath()` är en annan metod för att anropa QGIS-ikoner. Du hittar exempel på anrop till temaikoner på QGIS embedded images - Cheatsheet.

16.2.3 Gränssnitt för plugin i dialogrutan för alternativ

Du kan lägga till en anpassad flik för tilläggets alternativ i *Inställningar* ► *Alternativ*. Detta är att föredra framför att lägga till en specifik huvudmenypost för ditt tilläggs alternativ, eftersom det håller alla QGIS-applikationens inställningar och tilläggets inställningar på ett enda ställe som är lätt för användare att upptäcka och navigera.

Följande snippet kommer bara att lägga till en ny tom flik för plugin-inställningarna, redo för dig att fylla i alla alternativ och inställningar som är specifika för ditt plugin. Du kan dela upp följande klasser i olika filer. I det här exemplet lägger vi till två klasser i huvudfilen `mainPlugin.py`.

```
class MyPluginOptionsFactory(QgsOptionsWidgetFactory):

```

(continues on next page)

(fortsättning från föregående sida)

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

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

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


class ConfigOptionsPage(QgsOptionsPageWidget):

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

Slutligen lägger vi till importen och modifierar funktionen `__init__`:

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


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

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

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


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

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

Tips

Lägg till anpassade flikar i dialogen för lageregenskaper

Du kan tillämpa en liknande logik för att lägga till det anpassade alternativet för tillägget i dialogen med lageregenskaper med hjälp av klasserna `QgsMapLayerConfigWidgetFactory` och `QgsMapLayerConfigWidget`.

16.2.4 Bädda in anpassade widgetar för lager i lagerträdet

Förutom de vanliga lagersymbologielementen som visas bredvid eller under lagerposten i panelen *Layers* kan du lägga till egna widgetar som ger snabb åtkomst till vissa åtgärder som ofta används med ett lager (konfigurationsfiltrering, urval, stil, uppdatering av ett lager med en knappwidget, skapa en lagerbaserad tidsregulator eller bara visa extra lagerinformation i en etikett där, eller …). Dessa så kallade **Layer tree embedded widgets** görs tillgängliga via lagrets egenskaper *Legend*-fliken för enskilda lager.

Följande kodsnutt skapar en rullgardinsmeny i förklaringen som visar de lager-stilar som finns tillgängliga för lagret, så att du snabbt kan växla mellan de olika lager-stilarna.

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

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

        self.currentIndexChanged.connect(self.on_current_changed)

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

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

    def id(self):
        return "style"

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

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

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

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

Dra sedan `Layer style chooser` från *Available widgets* till *Used widgets* från ett visst lagers egenskapsflik *Legend* för att aktivera widgeten i lagerträdet. Inbäddade widgetar visas ALLTID högst upp i underobjekten i den tillhörande lagernoden.

Om du vill använda widgetarna från t.ex. ett plugin kan du lägga till dem så här:

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

16.3 IDE-inställningar för skrivning och debuggning av tillägg

Även om varje programmerare har sin föredragna IDE/Textredigerare, följer här några rekommendationer för att konfigurera populära IDE:er för att skriva och felsöka QGIS Python-tillägg.

16.3.1 Användbara tillägg för att skriva Python-tillägg

Vissa tillägg är praktiska när man skriver Python-tillägg. Från *Tillägg* ► *Hantera och installera tillägg…*, installera:

- *Återladdare för tillägg*: Detta gör att du kan ladda om ett plugin och dra nya ändringar utan att starta om QGIS.
- *Första hjälpen*: Detta lägger till en Python-konsol och en lokal felsökare för att inspektera variabler när ett undantag uppstår från ett plugin.

> **Varning**
>
> Trots våra ständiga ansträngningar kan det hända att information bortom denna linje inte uppdateras för QGIS 3.

16.3.2 En anteckning om hur du konfigurerar din IDE under Linux och Windows

På Linux är allt som vanligtvis behöver göras att lägga till QGIS-biblioteksplatserna i användarens miljövariabel `PYTHONPATH`. Under de flesta distributioner kan detta göras genom att redigera `~/.bashrc` eller `~/.bash-profile` med följande rad (testad på OpenSUSE Tumbleweed):

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

Spara filen och implementera miljöinställningarna med hjälp av följande shell-kommando:

```
source ~/.bashrc
```

På Windows måste du se till att du har samma miljöinställningar och använder samma bibliotek och tolk som QGIS. Det snabbaste sättet att göra detta är att ändra startbatchfilen för QGIS.

Om du använde OSGeo4W Installer kan du hitta den här under mappen `bin` i din OSGeo4W-installation. Leta efter något som `C:\OSGeo4W\bin\qgis-unstable.bat`.

16.3.3 Felsökning med hjälp av Pyscripter IDE (Windows)

För att använda Pyscripter IDE, här är vad du måste göra:

1. Gör en kopia av `qgis-unstable.bat` och döp om den till `pyscripter.bat`.
2. Öppna den i en editor. Och ta bort den sista raden, den som startar QGIS.
3. Lägg till en rad som pekar på den körbara filen för Pyscripter och lägg till kommandoradsargumentet som anger vilken version av Python som ska användas
4. Lägg också till argumentet som pekar på mappen där Pyscripter kan hitta Python dll som används av QGIS, du kan hitta den under bin-mappen i din OSGeoW-installation

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

5. När du nu dubbelklickar på denna batchfil kommer den att starta Pyscripter med rätt sökväg.

Eclipse är mer populärt än Pyscripter och är ett vanligt val bland utvecklare. I följande avsnitt förklarar vi hur du konfigurerar det för att utveckla och testa tillägg.

16.3.4 Felsökning med hjälp av Eclipse och PyDev

Installation

För att använda Eclipse måste du se till att du har installerat följande

- Eclipse
- Aptana Studio 3 Plugin eller PyDev
- QGIS 3.x
- Du kanske också vill installera **Remote Debug**, en QGIS plugin. För tillfället är det fortfarande experimentellt så aktivera  *Experimental tillägg* under *Tillägg* ► *Hantera och installera tillägg…* ► *Alternativ* i förväg.

För att förbereda din miljö för att använda Eclipse i Windows bör du också skapa en batchfil och använda den för att starta Eclipse:

1. Leta reda på mappen där `qgis_core.dll` finns. Normalt är detta `C:\OSGeo4W\apps\qgis\bin`, men om du har kompilerat ditt eget QGIS-program finns detta i din byggmapp i `output/bin/RelWithDebInfo`
2. Leta reda på din körbara fil `eclipse.exe`.
3. Skapa följande skript och använd det för att starta eclipse när du utvecklar QGIS-tillägg.

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

Konfigurera Eclipse

1. I Eclipse skapar du ett nytt projekt. Du kan välja *General Project* och länka dina riktiga källor senare, så det spelar egentligen ingen roll var du placerar det här projektet.

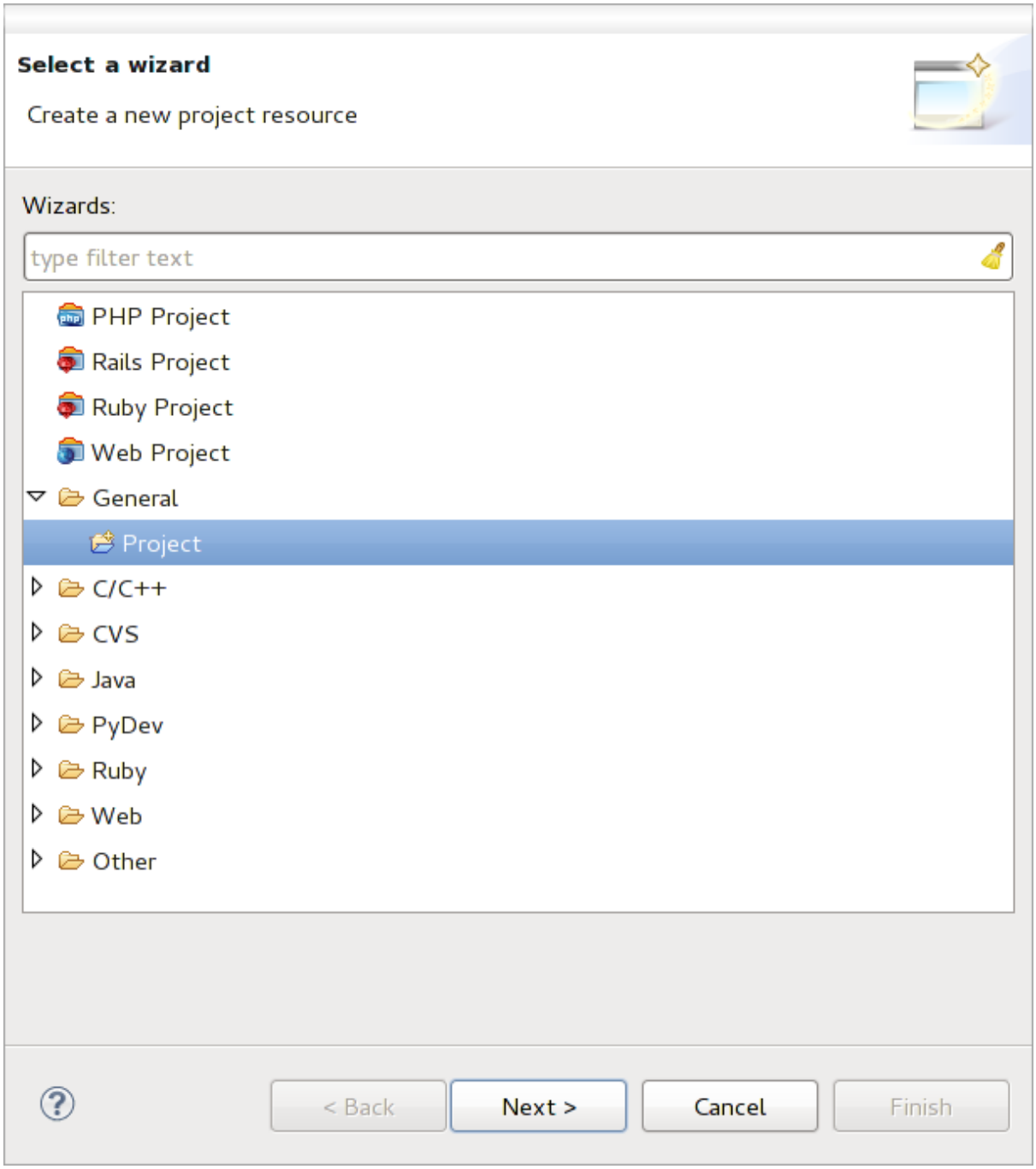


Fig. 16.1: Eclipse-projekt

2. Högerklicka på det nya projektet och välj *Ny* ► *Mapp*.
3. Klicka på *Avancerat* och välj *Länk till alternativ plats (Länkad mapp)*. Om du redan har källor som du vill felsöka väljer du dessa. Om du inte har det, skapa en mapp som det redan har förklarats.

I vyn *Project Explorer* dyker nu källträdet upp och du kan börja arbeta med koden. Du har redan syntaxmarkering och alla de andra kraftfulla IDE-verktygen tillgängliga.

Konfigurera felsökaren

För att få felsökaren att fungera:

1. Byt till Debug-perspektivet i Eclipse (*Window* ► *Open Perspective* ► *Other* ► *Debug*).
2. starta PyDev debug server genom att välja *PyDev* ► *Start Debug Server*.
3. Eclipse väntar nu på en anslutning från QGIS till sin debugserver och när QGIS ansluter till debugservern kommer den att tillåta den att styra pythonskripten. Det är precis vad vi installerade *Remote Debug* plugin för. Så starta QGIS om du inte redan gjort det och klicka på buggsymbolen.

Nu kan du ställa in en brytpunkt och så snart koden träffar den kommer exekveringen att stoppas och du kan inspektera det aktuella tillståndet för ditt plugin. (Brytpunkten är den gröna pricken i bilden nedan, ställ in en genom att dubbelklicka i det vita utrymmet till vänster om den rad du vill att brytpunkten ska ställas in).

```
88
89      def printProfile(self):
90          printer = QPrinter( QPrinter.HighResolution )
91          printer.setOutputFormat( QPrinter.PdfFormat )
92          printer.setPaperSize( QPrinter.A4 )
93          printer.setOrientation( QPrinter.Landscape )
94
95          printPreviewDlg = QPrintPreviewDialog( )
96          printPreviewDlg.paintRequested.connect( self.printRequested )
97
98          printPreviewDlg.exec_()
99
100     @pyqtSlot( QPrinter )
101     def printRequested( self, printer ):
102         self.webView.print_( printer )
```

Fig. 16.2: Brytpunkt

En mycket intressant sak som du kan använda dig av nu är debug-konsolen. Se till att exekveringen för närvarande är stoppad vid en brytpunkt innan du fortsätter.

1. Öppna konsolvyn (*Window* ► *Show view*). Den visar konsolen *Debug Server* som inte är särskilt intressant. Men det finns en knapp *Open Console* som låter dig byta till en mer intressant PyDev Debug Console.
2. Klicka på pilen bredvid knappen *Open Console* och välj *PyDev Console*. Ett fönster öppnas där du får frågan om vilken konsol du vill starta.
3. Välj *PyDev Debug Console*. Om den är gråtonad och säger åt dig att starta felsökaren och välja den giltiga ramen, se till att du har fjärrfelsökaren ansluten och för närvarande befinner dig på en brytpunkt.

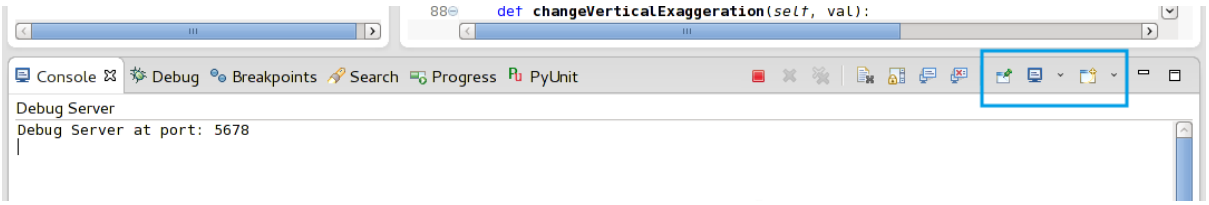


Fig. 16.3: PyDev felsökningskonsol

Du har nu en interaktiv konsol som låter dig testa alla kommandon från det aktuella sammanhanget. Du kan manipulera variabler eller göra API-anrop eller vad du vill.

Tips

Lite irriterande är att varje gång du anger ett kommando växlar konsolen tillbaka till Debug Server. För att stoppa detta beteende kan du klicka på knappen *Pin Console* när du är på Debug Server-sidan och den bör komma ihåg detta beslut åtminstone för den aktuella debug-sessionen.

Få eclipse att förstå API:et

En mycket praktisk funktion är att Eclipse faktiskt känner till QGIS API. Detta gör att den kan kontrollera din kod för stavfel. Men inte bara detta, det gör det också möjligt för Eclipse att hjälpa dig med autokomplettering från importen till API-anrop.

För att göra detta analyserar Eclipse QGIS-biblioteksfilerna och hämtar all information som finns där. Det enda du behöver göra är att tala om för Eclipse var biblioteken finns.

1. Klicka på *Fönster* ► *Inställningar* ► *PyDev* ► *Tolkprogram* ► *Python*.

   Du kommer att se din konfigurerade python-tolk i den övre delen av fönstret (för närvarande python2.7 för QGIS) och några flikar i den nedre delen. De flikar som är intressanta för oss är *Libraries* och *Forced Builtins*.

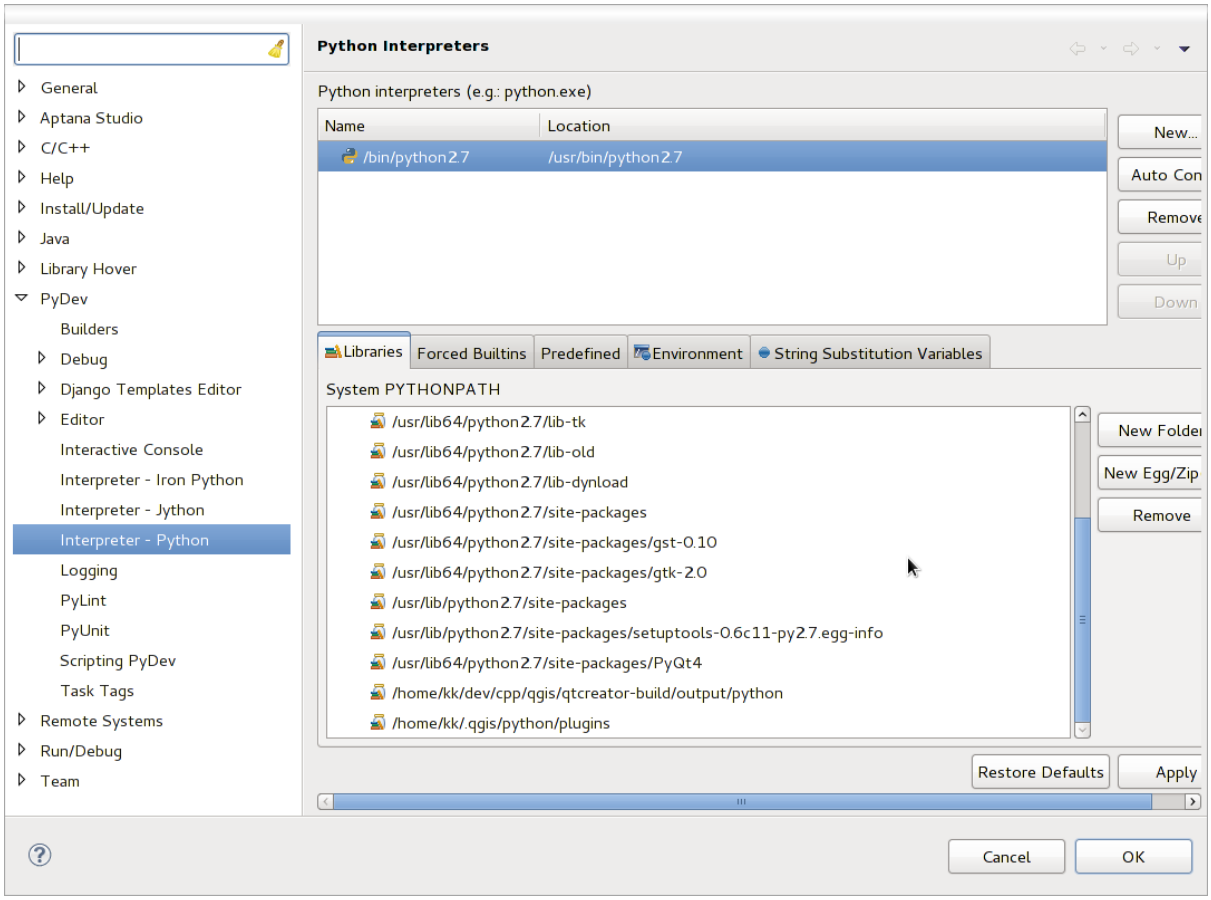


Fig. 16.4: PyDev felsökningskonsol

2. Öppna först fliken Libraries.
3. Add a New Folder och välj python-mappen i din QGIS-installation. Om du inte vet var den här mappen finns (det är inte tillägg-mappen):
   1. Öppna QGIS
   2. Starta en python-konsol
   3. Ange `qgis`

4. och tryck på Enter. Du kommer att se vilken QGIS-modul som används och dess sökväg.
5. Ta bort den efterföljande `/qgis/__init__.pyc` från den här sökvägen så har du den sökväg du letar efter.

4. Du bör också lägga till din tillägg-mapp här (den finns i `python/tillägg` under user profile-mappen).
5. Gå sedan till fliken *Forced Builtins*, klicka på *New…* och skriv in `qgis`. Detta kommer att få Eclipse att analysera QGIS API. Du vill förmodligen också att Eclipse ska känna till PyQt API. Lägg därför också till PyQt som forced builtin. Det bör förmodligen redan finnas i din biblioteksflik.
6. Klicka på *OK* och du är klar.

Observera

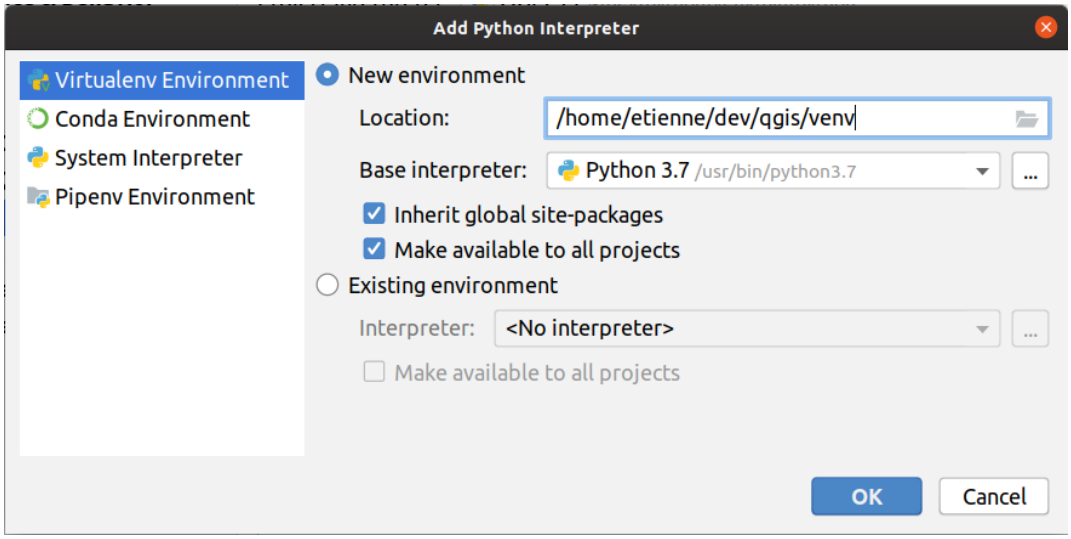
Varje gång QGIS API ändras (t.ex. om du kompilerar QGIS master och SIP-filen ändras), bör du gå tillbaka till den här sidan och klicka på *Apply*. Detta kommer att låta Eclipse analysera alla bibliotek igen.

16.3.5 Felsökning med PyCharm på Ubuntu med en kompilerad QGIS

PyCharm är en IDE för Python som utvecklats av JetBrains. Det finns en gratisversion som heter Community Edition och en betald version som heter Professional. Du kan ladda ner PyCharm på webbplatsen: https://www.jetbrains.com/pycharm/download

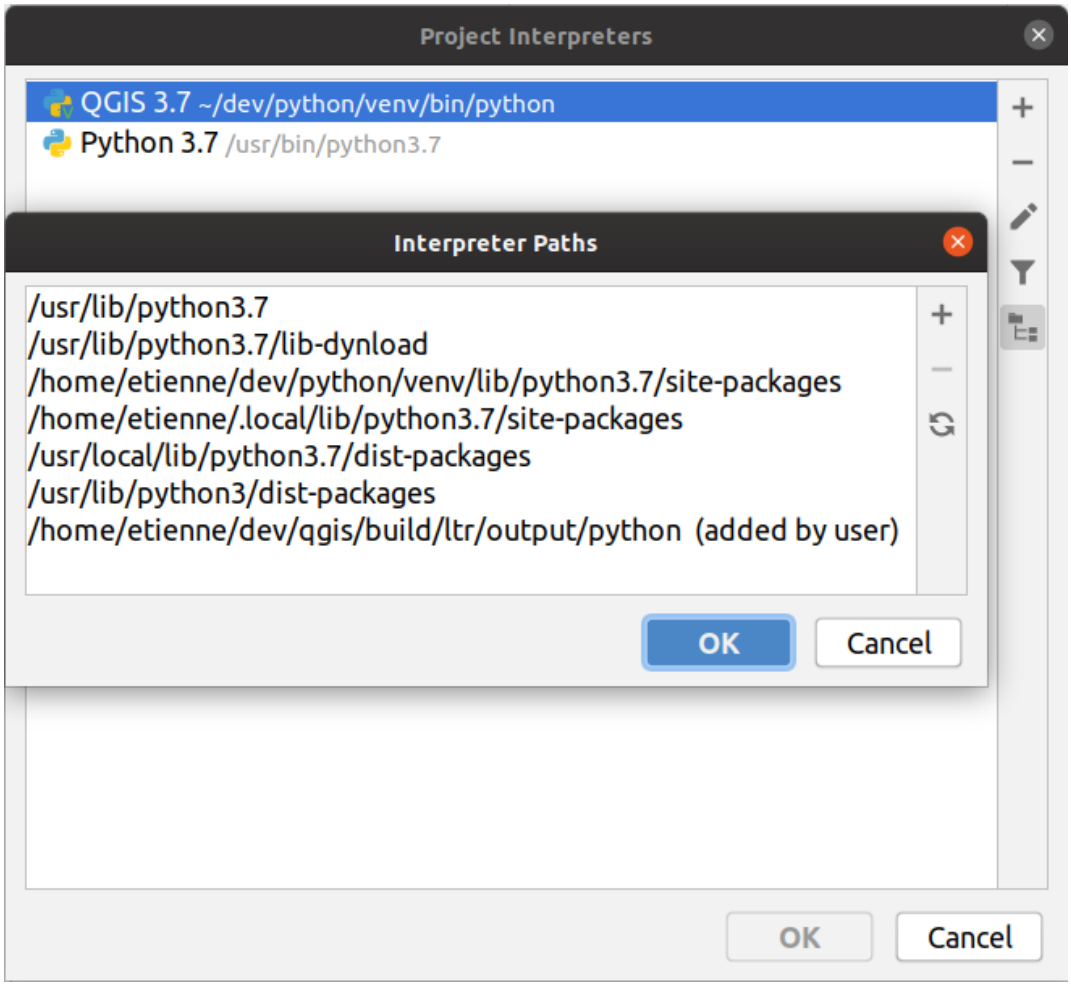
Vi antar att du har kompilerat QGIS på Ubuntu med den angivna byggkatalogen `~/dev/qgis/build/master`. Det är inte obligatoriskt att ha ett egenkompilerat QGIS, men endast detta har testats. Sökvägar måste anpassas.

1. I PyCharm, i din *Projektegenskaper*, *Project Interpreter*, kommer vi att skapa en virtuell Python-miljö som heter `QGIS`.
2. Klicka på den lilla kugghjulet och sedan på *Add*.
3. Välj *Virtualenv-miljö*.
4. Välj en generisk plats för alla dina Python-projekt, t.ex. `~/dev/qgis/venv` eftersom vi kommer att använda denna Python-tolk för alla våra tillägg.
5. Välj en Python 3-basinterpretator som finns tillgänglig på ditt system och kontrollera de två följande alternativen *Inherit global site-packages* och *Make available to all projects*.



1. Klicka på *OK*, gå tillbaka till den lilla växeln och klicka på *Visa alla*.
2. I det nya fönstret väljer du din nya tolk `QGIS` och klickar på den sista ikonen i den vertikala menyn *Visa sökvägar för den valda tolken.*

3. Slutligen lägger du till följande absoluta sökväg i listan `~/dev/qgis/build/master/output/python`.



1. Starta om PyCharm så kan du börja använda denna nya virtuella Python-miljö för alla dina tillägg.

PyCharm kommer att vara medveten om QGIS API och även om PyQt API om du använder Qt som tillhandahålls av QGIS som `from qgis.PyQt.QtCore import QDir`. Autokompletteringen bör fungera och PyCharm kan inspektera din kod.

I den professionella versionen av PyCharm fungerar fjärrfelsökning bra. För Community-utgåvan är fjärrfelsökning inte tillgänglig. Du kan bara ha tillgång till en lokal felsökare, vilket innebär att koden måste köras *inom* PyCharm (som skript eller unittest), inte i själva QGIS. För Python-kod som körs *i* QGIS kan du använda tillägget *First Aid* som nämns ovan.

16.3.6 Felsökning med hjälp av PDB

Om du inte använder en IDE som Eclipse eller PyCharm kan du felsöka med hjälp av PDB genom att följa dessa steg.

1. Lägg först till den här koden på den plats där du vill göra felsökningen

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

2. Kör sedan QGIS från kommandoraden.

   På Linux gör du det:

   ```
   $ ./Qgis
   ```

   På macOS gör du det:

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

3. Och när programmet når din brytpunkt kan du skriva i konsolen!

TODO:
Lägg till testinformation

16.4 Lansera ditt tillägg

När ditt tillägg är klart och du tror att tillägget kan vara till hjälp för vissa människor, tveka inte att ladda upp det till *Officiellt Python-tilläggsarkiv*. På den sidan kan du också hitta riktlinjer för paketering om hur du förbereder tillägget för att fungera bra med plugin-installationsprogrammet. Om du vill skapa ditt eget tilläggsarkiv kan du skapa en enkel XML-fil som listar tillägget och deras metadata.

Var särskilt noga med att följa följande anvisningar:

16.4.1 Metadata och namn

- undvika att använda ett namn som är alltför likt befintliga tillägg
- om ditt tillägg har en liknande funktionalitet som ett befintligt tillägg, förklara skillnaderna i fältet Om, så att användaren vet vilket som ska användas utan att behöva installera och testa det
- undvika att upprepa ”plugin” i namnet på själva tillägget
- använd beskrivningsfältet i metadata för en beskrivning på 1 rad, fältet Om för mer detaljerade instruktioner
- inkludera ett kodförvar, en buggspårare och en hemsida; detta kommer att förbättra möjligheten till samarbete avsevärt och kan göras mycket enkelt med någon av de tillgängliga webbinfrastrukturerna (GitHub, GitLab, Bitbucket, etc.)
- välj taggar med omsorg: undvik de oinformativa (t.ex. vector) och föredra de som redan används av andra (se plugin-webbplatsen)
- lägg till en riktig ikon, lämna inte standardikonen; se QGIS-gränssnittet för ett förslag på vilken stil som ska användas

16.4.2 Kod och hjälp

- inkludera inte genererade filer (ui_*.py, resources_rc.py, genererade hjälpfiler …) och värdelösa saker (t.ex. .gitignore) i repository
- lägga till tillägget i lämplig meny (Vector, Raster, Web, Database)
- när det är lämpligt (tillägg som utför analyser), överväg att lägga till tillägget som ett underplugin till Processing framework: detta gör det möjligt för användare att köra det i batch, att integrera det i mer komplexa arbetsflöden och befriar dig från bördan att utforma ett gränssnitt
- innehålla åtminstone minimal dokumentation och, om det är användbart för testning och förståelse, exempeldata.

16.4.3 Officiellt Python-tilläggsarkiv

Du kan hitta det *officiella* Python tilläggsarkivet på https://plugins.qgis.org/.

För att kunna använda det officiella arkivet måste du skaffa ett OSGEO ID från OSGEO webbportal.

När du har laddat upp din tillägg kommer den att godkännas av en medlem och du kommer att meddelas.

TODO:
Infoga en länk till styrdokumentet

Behörigheter

Dessa regler har implementerats i det officiella tilläggsarkivet:

- varje registrerad användare kan lägga till ett nytt tillägg
- *staff* -användare kan godkänna eller avvisa alla tilläggsversioner
- användare som har den särskilda behörigheten *tillägg.can_approve* får de versioner de laddar upp automatiskt godkända
- användare som har den speciella behörigheten *tillägg.can_approve* kan godkänna versioner som laddats upp av andra så länge de finns i listan över plugin *ägare*
- ett visst plugin kan raderas och redigeras endast av *anställda* användare och plugin *ägare*
- om en användare utan *tillägg.can_approve*-behörighet laddar upp en ny version, blir plugin-versionen automatiskt ogodkänd.

Förtroendehantering

Personal kan ge *trust* till utvalda plugin-skapare genom att ställa in *tillägg.can_approve*-behörighet via frontend-applikationen.

I plugin-detaljerna finns direktlänkar för att ge förtroende till plugin-skaparen eller plugin-ägarna*.

Validering

Tilläggets metadata importeras och valideras automatiskt från det komprimerade paketet när tillägget laddas upp.

Här är några valideringsregler som du bör känna till när du vill ladda upp ett plugin på det officiella förvaret:

1. namnet på huvudmappen som innehåller ditt plugin får endast innehålla ASCII-tecken (A-Z och a-z), siffror och tecknen underscore (_) och minus (-), och det får inte börja med en siffra
2. `metadata.txt` krävs
3. alla obligatoriska metadata som anges i *metadatatabell* måste finnas
4. metadatafältet *version* måste vara unikt
5. en licensfil måste inkluderas, sparad som `LICENSE` utan tillägg (dvs. inte `LICENSE.txt` till exempel)

Tilläggsstruktur

Enligt valideringsreglerna måste det komprimerade (.zip) paketet med ditt plugin ha en specifik struktur för att valideras som ett funktionellt plugin. Eftersom tillägget kommer att packas upp i användarens tilläggsmapp måste det ha en egen katalog i .zip-filen för att inte störa andra tillägg. Obligatoriska filer är: `metadata.txt`, `__init__.py` och `LICENSE`. Men det skulle vara trevligt att ha en `README` och naturligtvis en ikon för att representera tillägget. Nedan följer ett exempel på hur en `plugin.zip` kan se ut.

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

Det är möjligt att skapa tillägg i programmeringsspråket Python. I jämförelse med klassiska tillägg skrivna i C++ bör dessa vara enklare att skriva, förstå, underhålla och distribuera på grund av Python-språkets dynamiska karaktär.

Python-tillägg listas tillsammans med C++-tillägg i QGIS plugin-hanterare. De söks efter i `~/(UserProfile)/python/tillägg` och dessa sökvägar:

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

För definitioner av ~ och `(UserProfile)` se core_and_external_plugins.

Observera

Genom att ange *QGIS_PLUGINPATH* till en befintlig katalogsökväg kan du lägga till denna sökväg i listan över sökvägar som genomsöks efter tillägg.

KAPITEL 17

Skriva ett Processing-tillägg

Beroende på vilken typ av plugin du ska utveckla kan det vara ett bättre alternativ att lägga till dess funktionalitet som en Processing-algoritm (eller en uppsättning av dem). Det skulle ge en bättre integration i QGIS, ytterligare funktionalitet (eftersom den kan köras i komponenterna i Processing, t.ex. modelleraren eller batchbehandlingsgränssnittet) och en snabbare utvecklingstid (eftersom Processing kommer att ta en stor del av arbetet).

För att distribuera dessa algoritmer bör du skapa ett nytt tillägg som lägger till dem i Processing Toolbox. Tillägget ska innehålla en algoritmleverantör som måste registreras när tillägget instansieras.

17.1 Skapa från grunden

Om du vill skapa ett plugin från början som innehåller en algoritmleverantör kan du följa dessa steg med hjälp av Plugin Builder:

1. Installera tillägget **Tilläggsbyggare**
2. Skapa ett nytt plugin med hjälp av Plugin Builder. När Plugin Builder frågar dig om vilken mall du vill använda väljer du ”Processing provider”.
3. Det skapade tillägget innehåller en provider med en enda algoritm. Både providerfilen och algoritmfilen är fullständigt kommenterade och innehåller information om hur man ändrar providern och lägger till ytterligare algoritmer. Hänvisa till dem för mer information.

17.2 Uppdatering av ett tillägg

Om du vill lägga till ditt befintliga plugin till Processing måste du lägga till lite kod.

1. I din fil `metadata.txt` behöver du lägga till en variabel:

   ```
   hasProcessingProvider=yes
   ```

2. I Python-filen där ditt tillägg konfigureras med `initGui`-metoden måste du anpassa några rader så här:

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

class YourPluginName:

    def __init__(self):
        self.provider = None

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

    def initGui(self):
        self.initProcessing()

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

3. Du kan skapa en mapp `processing_provider` med tre filer i den:
   - `__init__.py` med ingenting i den. Detta är nödvändigt för att skapa ett giltigt Python-paket.
   - `provider.py` som skapar Processing-providern och exponerar dina algoritmer.

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

from .example_processing_algorithm import ExampleProcessingAlgorithm

class Provider(QgsProcessingProvider):

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

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

    def id(self) -> str:
        """The ID of your plugin, used for identifying the provider.

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

    def name(self) -> str:
        """The human friendly name of your plugin in Processing.

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

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

   - `example_processing_algorithm.py` som innehåller exempelalgoritmfilen. Kopiera/klistra in innehållet i skriptmallfilen och uppdatera den efter dina behov.

Du bör ha ett träd som liknar detta:

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

4. Nu kan du ladda om ditt plugin i QGIS och du bör se ditt exempelskript i verktygslådan Processing och modelleraren.

17.3 Implementering av anpassade bearbetningsalgoritmer

17.3.1 Skapa en anpassad algoritm

Här är ett enkelt exempel på en anpassad buffertalgoritm:

```
from qgis.core import (
    QgsProcessingAlgorithm,
    QgsProcessingParameterFeatureSource,
    QgsProcessingParameterNumber,
    QgsProcessingParameterFeatureSink,
    QgsFeatureSink,
)

class BufferAlgorithm(QgsProcessingAlgorithm):

    INPUT = 'INPUT'
    DISTANCE = 'DISTANCE'
    OUTPUT = 'OUTPUT'

    def initAlgorithm(self, config=None):
        self.addParameter(QgsProcessingParameterFeatureSource(self.INPUT, 'Input layer'))
        self.addParameter(QgsProcessingParameterNumber(self.DISTANCE, 'Buffer distance', defaultValue=100.0))
        self.addParameter(QgsProcessingParameterFeatureSink(self.OUTPUT, 'Output layer'))

    def processAlgorithm(self, parameters, context, feedback):
        source = self.parameterAsSource(parameters, self.INPUT, context)
        distance = self.parameterAsDouble(parameters, self.DISTANCE, context)
        (sink, dest_id) = self.parameterAsSink(parameters, self.OUTPUT, context,
                                               source.fields(), source.wkbType(), source.sourceCrs())

        for f in source.getFeatures():
            f.setGeometry(f.geometry().buffer(distance, 5))
            sink.addFeature(f, QgsFeatureSink.FastInsert)

        return {self.OUTPUT: dest_id}

    def name(self):
        return 'buffer'

    def displayName(self):
        return 'Buffer Features'
```

(continues on next page)

(fortsättning från föregående sida)

```
    def group(self):
        return 'Examples'

    def groupId(self):
        return 'examples'

    def createInstance(self):
        return BufferAlgorithm()
```

17.3.2 Anpassning av algoritmdialogrutan

Anpassade dialogrutor är särskilt användbara när du arbetar med kapslade eller dynamiska indata, när parametrar beror på externa datakällor som API:er (t.ex. dynamiskt ifyllda rullgardinsmenyer) eller när du behöver avancerad validering och anpassat layoutbeteende som inte stöds av standarddialogrutan för bearbetning. För att åsidosätta standardgränssnittet (t.ex. för komplexa parametertyper eller dynamisk logik), underordna `QgsProcessingAlgorithmDialogBase`. För att återge ditt anpassade användargränssnitt i standarddialogfönstret för bearbetning måste du anropa `self.setMainWidget(panel)`, där `panel` är en `QgsPanelWidget` som innehåller din anpassade layout. Detta säkerställer att ditt gränssnitt visas korrekt och interagerar korrekt med Processing-ramverket.

Här är ett exempel som integrerar signalhantering med hjälp av QTimer för avbouncad ingång:

```
from qgis.PyQt.QtCore import Qt, QT_VERSION_STR, QTimer
from qgis.core import (
    QgsProcessingAlgorithm,
    QgsProcessingContext,
    QgsProcessingFeedback,
    Qgis,
)
from qgis.PyQt.QtWidgets import QWidget, QVBoxLayout, QLineEdit
from qgis import gui, processing
from datetime import datetime
from typing import Dict, Optional
from osgeo import gdal

class CustomAlgorithmDialog(gui.QgsProcessingAlgorithmDialogBase):
    def __init__(
        self,
        algorithm: QgsProcessingAlgorithm,
        parent: Optional[QWidget] = None,
        title: Optional[str] = None,
    ):
        super().__init__(
            parent,
            flags=Qt.WindowFlags(),
            mode=gui.QgsProcessingAlgorithmDialogBase.DialogMode.Single,
        )
        self.context = QgsProcessingContext()
        self.setAlgorithm(algorithm)
        self.setModal(True)
        self.setWindowTitle(title or algorithm.displayName())

        self.panel = gui.QgsPanelWidget()
        layout = self.buildDialog()
        self.panel.setLayout(layout)
        self.setMainWidget(self.panel)

        self.cancelButton().clicked.connect(self.reject)
```

(continues on next page)

(fortsättning från föregående sida)

```
    def buildDialog(self) -> QVBoxLayout:
        layout = QVBoxLayout()

        self.input = QLineEdit()

        # Set up a debounced signal using QTimer
        self._update_timer = QTimer(self, singleShot=True)
        self._update_timer.timeout.connect(self._on_collection_id_ready)
        self.input.textChanged.connect(self._on_collection_id_changed)

        layout.addWidget(self.input)

        return layout

    def _on_collection_id_changed(self):
        self._update_timer.start(500)  # Debounce input

    def _on_collection_id_ready(self):
        self.pushInfo("Fetching metadata for collection ID…")

    def getParameters(self) -> Dict:
        try:
            return {'DISTANCE': float(self.input.text())}
        except ValueError:
            raise ValueError("Invalid buffer distance")

    def processingContext(self):
        return self.context

    def createFeedback(self):
        return QgsProcessingFeedback()

    def runAlgorithm(self):
        context = self.processingContext()
        feedback = self.createFeedback()
        params = self.getParameters()

        self.pushDebugInfo(f"QGIS version: {Qgis.QGIS_VERSION}")
        self.pushDebugInfo(f"QGIS code revision: {Qgis.QGIS_DEV_VERSION}")
        self.pushDebugInfo(f"Qt version: {QT_VERSION_STR}")
        self.pushDebugInfo(f"GDAL version: {gdal.VersionInfo('--version')}")
        self.pushCommandInfo(f"Algorithm started at: {datetime.now().
↪isoformat(timespec='seconds')}")
        self.pushCommandInfo(f"Algorithm '{self.algorithm().displayName()}'␣
↪starting…")
        self.pushCommandInfo("Input parameters:")
        for k, v in params.items():
            self.pushCommandInfo(f"  {k}: {v}")

        results = processing.run(self.algorithm(), params, context=context,␣
↪feedback=feedback)
        self.setResults(results)
        self.showLog()
```

För att starta den anpassade dialogen för en viss algoritm instansierar du helt enkelt `CustomAlgorithmDialog` med din algoritminstans och anropar `exec()`:

```
dlg = CustomAlgorithmDialog(BufferAlgorithm())
dlg.exec()
```

17.3.3 Hantera Qt-signaler

När du bygger reaktiva dialogrutor ska du hantera signalanslutningar noggrant. I mönstret ovan används en QTimer för att fördröja inmatning från textfältet, vilket förhindrar snabba upprepade anrop. Detta är särskilt användbart när du hämtar metadata eller uppdaterar användargränssnittselement baserat på användarens inmatning. Anslut alltid signaler en gång (vanligtvis i `__init__`) och använd `singleShot=True` för att säkerställa att sloten bara utlöses en gång efter en fördröjning.

KAPITEL 18

Använda tilläggslager

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

from qgis.PyQt.QtGui import QImage
```

Om ditt tillägg använder sina egna metoder för att rendera ett kartlager kan det bästa sättet att implementera det vara att skriva en egen lagertyp baserad på `QgsPluginLayer`.

18.1 Underklassning av QgsPluginLayer

Nedan visas ett exempel på en minimal implementering av QgsPluginLayer. Det är baserat på den ursprungliga koden för Watermark exempel plugin.

Den anpassade renderingen är den del av implementationen som definierar den faktiska ritningen på duken.

```
class WatermarkLayerRenderer(QgsMapLayerRenderer):

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

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

(continues on next page)

(fortsättning från föregående sida)

```
        return True

class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

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

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

    def setTransformContext(self, ct):
        pass

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

    def readXml(self, node, context):
        pass

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

Tilläggslagret kan läggas till i projektet och på canvas som vilket annat kartlager som helst:

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

När man laddar ett projekt som innehåller ett sådant lager behövs en fabriksklass:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

    def createLayer(self):
        return WatermarkPluginLayer()

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


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

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

KAPITEL 19

Bibliotek för nätverksanalys

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

Biblioteket för nätverksanalys kan användas för att:

- skapa en matematisk graf från geografiska data (polylinje-vektorlager)
- implementera grundläggande metoder från grafteori (för närvarande endast Dijkstras algoritm)

Biblioteket för nätverksanalys skapades genom att exportera grundläggande funktioner från RoadGraphs kärntillägg och nu kan du använda dess metoder i tillägg eller direkt från Python-konsolen.

19.1 Allmän information

Kortfattat kan ett typiskt användningsfall beskrivas som:

1. skapa graf från geodata (vanligtvis polylinjevektorlager)
2. analys av körgrafer
3. använda analysresultat (t.ex. visualisera dem)

19.2 Bygga upp en graf

Det första du behöver göra — är att förbereda indata, det vill säga att konvertera ett vektorlager till en graf. Alla ytterligare åtgärder kommer att använda denna graf, inte lagret.

Som källa kan vi använda vilket polylinjevektorlager som helst. Polylinjernas noder blir grafvertexer och polylinjernas segment blir grafkanter. Om flera noder har samma koordinater är de samma grafvertex. Två linjer som har en gemensam nod blir alltså kopplade till varandra.

Under skapandet av grafen är det dessutom möjligt att ”fixera” (”knyta”) ytterligare ett antal punkter till det ingående vektorlagret. För varje ytterligare punkt hittas en matchning — den närmaste grafvertexen eller den närmaste grafkanten. I det senare fallet delas kanten och ett nytt vertex läggs till.

Vektorlagrets attribut och längden på en kant kan användas som egenskaper för en kant.

Konvertering från ett vektorlager till grafen görs med hjälp av programmeringsmönstret Builder. En graf konstrueras med hjälp av en så kallad Director. Det finns bara en Director för tillfället: `QgsVectorLayerDirector`. Director anger de grundläggande inställningar som ska användas för att konstruera en graf från ett linjevektorlager, som används av byggaren för att skapa grafen. För närvarande finns det, precis som i fallet med director, bara en byggare: `QgsGraphBuilder`, som skapar `QgsGraph`-objekt. Du kanske vill implementera dina egna byggare som bygger en graf som är kompatibel med sådana bibliotek som BGL eller NetworkX.

För att beräkna kantegenskaper används programmeringsmönstret strategy. För närvarande finns endast `QgsNetworkDistanceStrategy` strategi (som tar hänsyn till ruttens längd) och `QgsNetworkSpeedStrategy` (som även tar hänsyn till hastigheten) tillgängliga. Du kan implementera din egen strategi som kommer att använda alla nödvändiga parametrar. Tillägget RoadGraph använder t.ex. en strategi som beräknar restiden med hjälp av kantlängd och hastighetsvärde från attribut.

Det är dags att dyka in i processen.

För att kunna använda det här biblioteket måste vi först importera analysmodulen

```
from qgis.analysis import *
```

Sedan några exempel på hur man skapar en direktör

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)

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

För att konstruera en director måste vi skicka ett vektorlager som kommer att användas som källa för grafstrukturen och information om tillåten rörelse på varje vägsegment (enkelriktad eller dubbelriktad rörelse, direkt eller omvänd riktning). Samtalet ser ut så här

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

Och här är en fullständig lista över vad dessa parametrar betyder:

- `vectorLayer` — vektorlager som används för att bygga grafen

- `directionFieldId` — index för attributtabellens fält, där information om vägars riktning lagras. Om `-1`, använd inte denna information alls. Ett heltal.
- `directDirectionValue` — fältvärde för vägar med direkt riktning (förflyttning från första linjepunkten till den sista). En sträng.
- `reverseDirectionValue` — fältvärde för vägar med omvänd riktning (förflyttning från sista linjepunkten till den första). En sträng.
- `bothDirectionValue` — fältvärde för dubbelriktade vägar (för sådana vägar kan vi flytta från första punkten till sista och från sista till första). En sträng.
- `defaultDirection` — standard vägriktning. Detta värde kommer att användas för de vägar där fältet `directionFieldId` inte är inställt eller har ett annat värde än något av de tre värden som anges ovan. Möjliga värden är:
  - `QgsVectorLayerDirector.DirectionForward` — Enkelriktad direkt
  - `QgsVectorLayerDirector.DirectionBackward` — Enkelriktad backning
  - `QgsVectorLayerDirector.DirectionBoth` — Tvåvägs

Det är då nödvändigt att skapa en strategi för att beräkna kantegenskaper

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

Och berätta för chefen om denna strategi

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

Nu kan vi använda byggaren, som kommer att skapa grafen. Klassens `QgsGraphBuilder` konstruktor tar flera argument:

- `crs` — koordinatreferenssystem att använda. Obligatoriskt argument.
- `otfEnabled` — använd ”on the fly” reprojection eller inte. Som standard `True` (använd OTF).
- `topologyTolerance` — topologisk tolerans. Standardvärdet är 0.
- `ellipsoidID` — ellipsoid att använda. Som standard ”WGS84”.

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

Vi kan också definiera flera punkter som kommer att användas i analysen. Till exempel

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

Nu är allt på plats så att vi kan bygga grafen och ”knyta” dessa punkter till den

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

Att bygga grafen kan ta lite tid (vilket beror på antalet funktioner i ett lager och lagrets storlek). `tiedPoints` är en lista med koordinater för ”bundna” punkter. När byggprocessen är klar kan vi hämta grafen och använda den för analysen

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

Med följande kod kan vi få fram toppunktsindexen för våra punkter

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

19.3 Grafisk analys

Nätverksanalys används för att hitta svar på två frågor: vilka toppar som är anslutna och hur man hittar den kortaste vägen. För att lösa dessa problem tillhandahåller biblioteket för nätverksanalys Dijkstras algoritm.

Dijkstras algoritm hittar den kortaste vägen från en av grafens hörnpunkter till alla de andra och värdena på optimeringsparametrarna. Resultaten kan representeras som ett träd med kortaste vägen.

Trädet med den kortaste vägen är en riktad viktad graf (eller mer exakt ett träd) med följande egenskaper:

- endast ett toppunkt har inga inkommande kanter - trädets rot
- alla andra toppar har bara en inkommande kant
- om toppunkt B kan nås från toppunkt A, så är vägen från A till B den enda tillgängliga vägen och den är optimal (kortast) i denna graf

För att få trädet med den kortaste vägen används metoderna `shortestTree()` och `dijkstra()` i klassen `QgsGraphAnalyzer`. Det rekommenderas att använda metoden `dijkstra()` eftersom den är snabbare och använder minnet mer effektivt.

Metoden `shortestTree()` är användbar när du vill gå runt i trädet med den kortaste vägen. Den skapar alltid ett nytt grafobjekt (QgsGraph) och accepterar tre variabler:

- `source` — ingångsgraf
- `startVertexIdx` — index för punkten på trädet (trädets rot)
- `criterionNum` — antal edge-egenskaper som ska användas (börjar från 0).

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

Metoden `dijkstra()` har samma argument, men returnerar en tupel av arrayer:

- I den första arrayen innehåller element *n* index för den inkommande kanten eller -1 om det inte finns några inkommande kanter.
- I den andra arrayen innehåller elementet *n* avståndet från trädets rot till toppunkt *n* eller DOUBLE_MAX om toppunkt *n* inte kan nås från roten.

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

Här är en mycket enkel kod för att visa trädet med den kortaste vägen med hjälp av grafen som skapats med metoden `shortestTree()` (välj linjestringskiktet i panelen *Layers* och ersätt koordinaterna med dina egna).

> **Varning**
>
> Använd denna kod endast som ett exempel, den skapar många `QgsRubberBand`-objekt och kan vara långsam på stora datamängder.

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

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

(continues on next page)

(fortsättning från föregående sida)

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

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

i = 0
while (i < tree.edgeCount()):
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
  rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
  i = i + 1
```

Samma sak men med metoden `dijkstra()`

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

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

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

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
  rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())
```

19.3.1 Hitta de kortaste vägarna

För att hitta den optimala vägen mellan två punkter används följande tillvägagångssätt. Båda punkterna (start A och slut B) är ”bundna” till grafen när den byggs. Sedan använder vi metoden `shortestTree()` eller `dijkstra()` för att bygga trädet med den kortaste vägen med roten i startpunkten A. I samma träd hittar vi också slutpunkten B och börjar gå genom trädet från punkt B till punkt A. Hela algoritmen kan skrivas som:

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

Vid denna punkt har vi vägen, i form av den inverterade listan över vertex (vertex listas i omvänd ordning från slutpunkt till startpunkt) som kommer att besökas under resan med denna väg.

Här är exempelkoden för QGIS Python Console (du kan behöva ladda och välja ett linjestringslager i TOC och ersätta koordinaterna i koden med dina) som använder metoden `shortestTree()`

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(fortsättning från föregående sida)

```
    route.insert(0, tree.vertex(edge.fromVertex()).point())
    idxEnd = edge.fromVertex()

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

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

Och här är samma exempel men med metoden `dijkstra()`

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

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

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

builder = QgsGraphBuilder(vectorLayer.sourceCrs())

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

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

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

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

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

# Total cost
cost = costs[idxEnd]

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

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

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

# This may require coordinate transformation if project's CRS
```

(continues on next page)

(fortsättning från föregående sida)

```
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

19.3.2 Tillgängliga områden

Tillgänglighetsområdet för toppunkt A är den delmängd av grafens toppunkter som är tillgängliga från toppunkt A och där kostnaden för vägarna från A till dessa toppunkter inte är större än ett visst värde.

Detta kan tydliggöras med följande exempel: ”Det finns en brandstation. Vilka delar av staden kan en brandbil nå på 5 minuter? 10 minuter? 15 minuter?”. Svaren på dessa frågor är brandstationens tillgänglighetsområden.

För att hitta områden med tillgänglighet kan vi använda metoden `dijkstra()` i klassen `QgsGraphAnalyzer`. Det räcker att jämföra elementen i kostnadsarrayen med ett fördefinierat värde. Om cost[i] är mindre än eller lika med ett fördefinierat värde, ligger vertex i inom tillgänglighetsområdet, annars ligger det utanför.

Ett svårare problem är att få fram gränserna för tillgänglighetsområdet. Den nedre gränsen är den uppsättning vertex som fortfarande är tillgängliga, och den övre gränsen är den uppsättning vertex som inte är tillgängliga. I själva verket är detta enkelt: det är tillgänglighetsgränsen baserad på kanterna i trädet med den kortaste vägen för vilken källvertexen för kanten är tillgänglig och målvertexen för kanten inte är det.

Här följer ett exempel

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


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

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

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

idStart = graph.findVertex(tStart)

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

upperBound = []
r = 1500.0
i = 0
tree.reverse()

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

```

(continues on next page)

(fortsättning från föregående sida)

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

KAPITEL 20

QGIS Server och Python

20.1 Introduktion

Om du vill veta mer om QGIS Server kan du läsa QGIS-Server-manual.

QGIS Server är tre olika saker:

1. QGIS Server-bibliotek: ett bibliotek som tillhandahåller ett API för att skapa OGC-webbtjänster
2. QGIS Server FCGI: en binär FCGI-applikation `qgis_mapserv.fcgi` som tillsammans med en webbserver implementerar en uppsättning OGC-tjänster (WMS, WFS, WCS etc.) och OGC API:er (WFS3/OAPIF)
3. QGIS Development Server: en binär applikation för utvecklingsserver `qgis_mapserver` som implementerar en uppsättning OGC-tjänster (WMS, WFS, WCS etc.) och OGC API:er (WFS3/OAPIF)

Detta kapitel i kokboken fokuserar på det första ämnet och genom att förklara användningen av QGIS Server API visar det hur det är möjligt att använda Python för att utöka, förbättra eller anpassa serverns beteende eller hur man använder QGIS Server API för att bädda in QGIS Server i ett annat program.

Det finns några olika sätt att ändra QGIS Servers beteende eller utöka dess kapacitet för att erbjuda nya anpassade tjänster eller API:er, det här är de viktigaste scenarierna du kan ställas inför:

- EMBEDDING → Använd QGIS Server API från ett annat Python-program
- STANDALONE → Kör QGIS Server som en fristående WSGI/HTTP-tjänst
- FILTERS → Förbättra/anpassa QGIS Server med filtertillägg
- SERVICES → Lägg till en ny *SERVICE*
- OGC APIs → Lägg till ett nytt *OGC API*

Inbäddning och fristående applikationer kräver att du använder QGIS Server Python API direkt från ett annat Python-skript eller en annan applikation. De återstående alternativen är bättre lämpade för när du vill lägga till anpassade funktioner i en binär standardapplikation för QGIS Server (FCGI eller utvecklingsserver): i det här fallet måste du skriva ett Python-tillägg för serverapplikationen och registrera dina anpassade filter, tjänster eller API:er.

20.2 Grunderna i server-API:et

De grundläggande klasser som ingår i en typisk QGIS Server-applikation är:

- `QgsServer` serverinstansen (vanligtvis en enda instans under hela programmets livslängd)
- `QgsServerRequest` objektet för begäran (återskapas vanligtvis vid varje begäran)
- `QgsServer.handleRequest(request, response)` behandlar begäran och fyller i svaret

Arbetsflödet för QGIS Server FCGI eller utvecklingsserver kan sammanfattas enligt följande:

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

Inuti metoden `QgsServer.handleRequest(request, response)` anropas filtertilläggets callbacks och `QgsServerRequest` och `QgsServerResponse` görs tillgängliga för tilläggen genom klassen `QgsServerInterface`.

> **Varning**
>
> QGIS-serverklasser är inte trådsäkra, du bör alltid använda en multiprocessmodell eller containrar när du bygger skalbara applikationer baserade på QGIS Server API.

20.3 Fristående eller inbäddad

För fristående serverapplikationer eller inbäddning måste du använda de ovan nämnda serverklasserna direkt och förpacka dem i en webbserverimplementation som hanterar alla HTTP-protokollinteraktioner med klienten.

Här följer ett minimalt exempel på användning av QGIS Server API (utan HTTP-delen):

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

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

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

# Create a response objects
response = QgsBufferServerResponse()

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

print(response.headers())
```

(continues on next page)

(fortsättning från föregående sida)

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

app.exitQgis()
```

Här är ett komplett fristående applikationsexempel som utvecklats för kontinuerlig integrationstestning på QGIS källkodsarkiv, det visar en bred uppsättning olika tilläggsfilter och autentiseringsscheman (inte avsedda för produktion eftersom de endast utvecklades för teständamål men ändå intressanta för inlärning): qgis_wrapped_server.py

20.4 Tillägg för server

Python-tillägg för server laddas en gång när QGIS Server-programmet startar och kan användas för att registrera filter, tjänster eller API:er.

Strukturen för ett servertillägg är mycket lik dess motsvarighet på skrivbordet, ett `QgsServerInterface`-objekt görs tillgängligt för tilläggen och tillägget kan registrera ett eller flera anpassade filter, tjänster eller API:er till motsvarande register genom att använda en av de metoder som exponeras av servergränssnittet.

20.4.1 Tillägg för serverfilter

Filter finns i tre olika varianter och de kan instansieras genom att underklassa en av klasserna nedan och anropa motsvarande metod i `QgsServerInterface`:

Filtertyp	Basklass	Registrering av QgsServerInterface
I/O	`QgsServerFilter`	`registerFilter()`
Åtkomstkontroll	`QgsAccessControlFilter`	`registerAccessControl()`
Cache	`QgsServerCacheFilter`	`registerServerCache()`

I/O-filter

I/O-filter kan modifiera serverns in- och utdata (begäran och svar) för kärntjänsterna (WMS, WFS etc.), vilket gör det möjligt att manipulera tjänsternas arbetsflöde på alla sätt. Det är t.ex. möjligt att begränsa åtkomsten till utvalda lager, att injicera en XSL-stilmall i XML-svaret, att lägga till en vattenstämpel i en genererad WMS-bild och så vidare.

Från den här punkten kan det vara bra att snabbt titta på server tillägg API docs.

Varje filter ska implementera minst en av tre callbacks:

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

Alla filter har tillgång till request/response-objektet (`QgsRequestHandler`) och kan manipulera alla dess egenskaper (input/output) och skapa undantag (men på ett ganska speciellt sätt som vi ska se nedan).

Alla dessa metoder returnerar ett booleanskt värde som anger om anropet ska spridas till efterföljande filter. Om någon av dessa metoder returnerar `False` så stoppas kedjan, annars kommer anropet att spridas till nästa filter.

Här följer en pseudokod som visar hur servern hanterar en typisk förfrågan och när filtrets callbacks anropas:

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

    if there is not a response:
```

(continues on next page)

(fortsättning från föregående sida)

```
        if SERVICE is WMS/WFS/WCS:
            create WMS/WFS/WCS service
            call service's executeRequest
                possibly call onSendResponse for each chunk of bytes
                sent to the client by a streaming services (WFS)
        call onResponseComplete
    request handler sends the response to the client
```

I följande stycken beskrivs de tillgängliga callbacks i detalj.

onRequestReady

Detta kallas när begäran är klar: inkommande URL och data har analyserats och innan de går in i kärntjänsterna (WMS, WFS etc.) växlar, detta är den punkt där du kan manipulera inmatningen och utföra åtgärder som:

- autentisering/auktorisering
- omdirigeringar
- lägga till/ta bort vissa parametrar (t.ex. typnamn)
- göra undantag

Du kan till och med ersätta en kärntjänst helt och hållet genom att ändra parametern **SERVICE** och därmed kringgå kärntjänsten helt och hållet (vilket dock inte är särskilt meningsfullt).

onSendResponse

Detta anropas närhelst någon del av utdata spolas från svarsbufferten (dvs. till **FCGI** `stdout` om fcgi-servern används) och därifrån till klienten. Detta inträffar när stort innehåll strömmas (som WFS GetFeature). I detta fall kan `onSendResponse()` anropas flera gånger.

Observera att om svaret inte är streamat, kommer `onSendResponse()` inte att anropas alls.

I samtliga fall kommer den sista (eller unika) delen att skickas till klienten efter ett anrop till `onResponseComplete()`.

Att returnera `False` kommer att förhindra att data skickas till klienten. Detta är önskvärt när ett tillägg vill samla in alla delar från ett svar och undersöka eller ändra svaret i `onResponseComplete()`.

onResponseComplete

Detta anropas en gång när kärntjänsterna (om de har träffats) har avslutat sin process och begäran är klar att skickas till klienten. Som diskuterats ovan kommer den här metoden att anropas innan den sista (eller unika) delen av data skickas till klienten. För strömningstjänster kan flera anrop till `onSendResponse()` ha anropats.

`onResponseComplete()` är den perfekta platsen för att implementera nya tjänster (WPS eller anpassade tjänster) och för att utföra direkt manipulation av utdata som kommer från kärntjänster (till exempel för att lägga till en vattenstämpel på en WMS-bild).

Observera att om du returnerar `False` förhindras nästa tillägg att utföra `onResponseComplete()`, men i vilket fall som helst förhindras att svaret skickas till klienten.

Utlösa undantag från ett tillägg

Det återstår fortfarande en del arbete med detta ämne: den nuvarande implementeringen kan skilja mellan hanterade och ohanterade undantag genom att ställa in en `QgsRequestHandler`-egenskap till en instans av QgsMapServiceException, på så sätt kan den huvudsakliga C++-koden fånga hanterade pythonundantag och ignorera ohanterade undantag (eller bättre: logga dem).

Detta tillvägagångssätt fungerar i princip men det är inte särskilt ”pythoniskt”: ett bättre tillvägagångssätt skulle vara att skapa undantag från pythonkod och se dem bubbla upp till C++-loopen för att hanteras där.

Skriva ett servertillägg

Ett servertillägg är ett standard QGIS Python-tillägg enligt beskrivningen i *Utveckla Python-tillägg*, som bara tillhandahåller ett ytterligare (eller alternativt) gränssnitt: ett typiskt QGIS-skrivbordsplugin har tillgång till QGIS-applikationen via `QgisInterface`, medan ett servertillägg endast har tillgång till `QgsServerInterface` när det körs i QGIS Server-applikationskontext.

För att göra QGIS Server medveten om att ett tillägg har ett servergränssnitt behövs en speciell metadatapost (i `metadata.txt`):

```
server=True
```

> **Viktigt**
>
> Endast tillägg som har metadatauppsättningen `server=True` kommer att laddas och köras av QGIS Server.

Exempeltillägget qgis3-server-vagrant som diskuteras här (med många fler) finns tillgängligt på github, några servertillägg publiceras också i det officiella QGIS tillägg repository.

Tilläggsfiler

Här är katalogstrukturen för vårt exempel på servertillägg.

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

__init__.py

Denna fil krävs av Pythons importsystem. QGIS Server kräver också att denna fil innehåller en `serverClassFactory()`-funktion, som anropas när tillägget laddas in i QGIS Server när servern startar. Den tar emot en referens till en instans av `QgsServerInterface` och måste returnera en instans av plugin-programmets klass. Så här ser exempelpluginet `__init__.py` ut:

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

HelloServer.py

Det är här det magiska händer och det är så här det magiska ser ut: (t.ex. `HelloServer.py`)

Ett servertillägg består vanligtvis av en eller flera callbacks som packas in i instanser av en `QgsServerFilter`.

Varje `QgsServerFilter` implementerar en eller flera av följande callbacks:

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

I följande exempel implementeras ett minimalt filter som skriver ut *HelloServer!* om parametern **SERVICE** är lika med ”HELLO”:

```
class HelloFilter(QgsServerFilter):

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

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

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

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

Filtren måste registreras i **serverIface** som i följande exempel:

```
class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(serverIface), 100)
```

Den andra parametern i `registerFilter()` anger en prioritet som definierar ordningen för callbacks med samma namn (den med lägre prioritet anropas först).

Genom att använda de tre callbacks kan tillägg manipulera ingången och/eller utgången från servern på många olika sätt. I varje ögonblick har plugin-instansen tillgång till `QgsRequestHandler` genom `QgsServerInterface`. Klassen `QgsRequestHandler` har många metoder som kan användas för att ändra ingångsparametrarna innan de går in i serverns kärnbearbetning (genom att använda `requestReady()`) eller efter att begäran har bearbetats av kärntjänsterna (genom att använda `sendResponse()`).

Följande exempel täcker några vanliga användningsfall:

Modifiera inmatningen

Exempeltillägget innehåller ett testexempel som ändrar inmatningsparametrar som kommer från frågesträngen, i det här exemplet injiceras en ny parameter i (redan analyserad) `parameterMap`, den här parametern är sedan synlig av kärntjänster (WMS etc.), i slutet av kärntjänstbearbetningen kontrollerar vi att parametern fortfarande finns kvar:

```
class ParamsFilter(QgsServerFilter):

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

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

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

Detta är ett utdrag av vad du ser i loggfilen:

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

På den markerade raden anger strängen ”SUCCESS” att tillägget klarade testet.

Samma teknik kan utnyttjas för att använda en anpassad tjänst i stället för en kärntjänst: du kan t.ex. hoppa över en **WFS SERVICE**-begäran eller någon annan kärntjänstbegäran bara genom att ändra **SERVICE**-parametern till något annat och kärntjänsten kommer att hoppas över. Sedan kan du injicera dina anpassade resultat i utdata och skicka dem till klienten (detta förklaras nedan).

Tips

Om du verkligen vill implementera en anpassad tjänst rekommenderas att du underklassar `QgsService` och registrerar din tjänst på `registerFilter()` genom att anropa dess `registerService(service)`

Modifiera eller byta ut utmatningen

Exemplet med vattenstämpelfiltret visar hur man ersätter WMS-utdata med en ny bild som erhålls genom att lägga till en vattenstämpelbild ovanpå den WMS-bild som genereras av WMS-kärntjänsten:

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

class WatermarkFilter(QgsServerFilter):

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

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (params.get('SERVICE').upper() == 'WMS' \
                and params.get('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.onResponseComplete: image ready %s" % request.parameter("FORMAT"))
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG" if "png" in request.parameter("FORMAT") else "JPG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
        return True
```

I det här exemplet kontrolleras parametervärdet **SERVICE** och om den inkommande begäran är en **WMS GETMAP** och inga undantag har ställts in av ett tidigare exekverat plugin eller av kärntjänsten (WMS i det här fallet), hämtas den WMS-genererade bilden från utmatningsbufferten och vattenstämpelbilden läggs till. Det sista steget är att rensa utmatningsbufferten och ersätta den med den nyligen genererade bilden. Observera att i en verklig situation bör vi också kontrollera den begärda bildtypen i stället för att bara stödja PNG eller JPG.

Filter för åtkomstkontroll

Filter för åtkomstkontroll ger utvecklaren en finkornig kontroll över vilka lager, funktioner och attribut som kan nås, följande callbacks kan implementeras i ett filter för åtkomstkontroll:

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

- `cacheKey()`

Tilläggsfiler

Här är katalogstrukturen för vårt exempel på tillägg:

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

__init__.py

Den här filen krävs av Pythons importsystem. Som för alla QGIS-servertillägg innehåller denna fil en `serverClassFactory()`-funktion, som anropas när tillägget laddas in i QGIS Server vid uppstart. Den tar emot en referens till en instans av `QgsServerInterface` och måste returnera en instans av ditt tilläggs klass. Så här ser exempelpluginet `__init__.py` ut:

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

AccessControl.py

```
class AccessControlFilter(QgsAccessControlFilter):

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

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

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

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

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

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

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

class AccessControlServer:

   def __init__(self, serverIface):
```

(continues on next page)

(fortsättning från föregående sida)

```
32      """ Register AccessControlFilter """
33      serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

Detta exempel ger full åtkomst för alla.

Det är tilläggets uppgift att veta vem som är inloggad.

För alla dessa metoder har vi argumentet lager på lager för att kunna anpassa begränsningen per lager.

layerFilterExpression

Används för att lägga till ett uttryck för att begränsa resultaten.

Till exempel för att begränsa till funktioner där attributet `role` är lika med `user`.

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

layerFilterSubsetString

Samma som föregående men använd `SubsetString` (exekveras i databasen)

Till exempel för att begränsa till funktioner där attributet `role` är lika med `user`.

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

layerPermissions

Begränsa åtkomsten till lagret.

Returnera ett objekt av typen `LayerPermissions()`, som har egenskaperna:

- `canRead` för att se den i `GetCapabilities` och ha läsbehörighet.
- `canInsert` för att kunna infoga en ny funktion.
- `canUpdate` för att kunna uppdatera en funktion.
- `canDelete` för att kunna ta bort en funktion.

Till exempel för att begränsa allt till endast läsrättigheter:

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

authorizedLayerAttributes

Används för att begränsa synligheten för en specifik delmängd av attributet.

Argumentet attribute returnerar den aktuella uppsättningen synliga attribut.

Till exempel för att dölja attributet `role`:

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

allowToEdit

Detta används för att begränsa redigeringen till en delmängd av funktionerna.

Det används i protokollet `WFS-Transaction`.

Till exempel för att bara kunna redigera funktioner som har attributet `role` med värdet `user`:

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

cacheKey

QGIS Server upprätthåller en cache av funktionerna och om du vill ha en cache per roll kan du returnera rollen i den här metoden. Eller returnera `None` för att helt inaktivera cacheminnet.

20.4.2 Anpassade tjänster

I QGIS Server implementeras kärntjänster som WMS, WFS och WCS som underklasser till `QgsService`.

Om du vill implementera en ny tjänst som körs när frågesträngparametern `SERVICE` matchar tjänstens namn kan du implementera din egen `QgsService` och registrera tjänsten på `serviceRegistry()` genom att anropa dess `registerService(service)`.

Här är ett exempel på en anpassad tjänst med namnet `CUSTOM`:

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

class CustomServiceService(QgsService):

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

    def name(self):
        return "CUSTOM"

    def version(self):
        return "1.0.0"

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


class CustomService():

```

(continues on next page)

(fortsättning från föregående sida)

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

20.4.3 Anpassade API:er

I QGIS Server implementeras centrala OGC API:er som OAPIF (även känt som WFS3) som samlingar av `QgsServerOgcApiHandler` subklasser som är registrerade till en instans av `QgsServerOgcApi` (eller dess överordnade klass `QgsServerApi`).

För att implementera ett nytt API som kommer att köras när webbadressens sökväg matchar en viss URL kan du implementera dina egna `QgsServerOgcApiHandler`-instanser, lägga till dem i en `QgsServerOgcApi` och registrera API:et på `serviceRegistry()` genom att anropa dess `registerApi(api)`.

Här är ett exempel på ett anpassat API som kommer att köras när URL:en innehåller `/customapi`:

```
import json
import os

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

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


class CustomApiHandler(QgsServerOgcApiHandler):

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

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

    def operationId(self):
        return "CustomApiXYCircle"

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

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

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

    def linkType(self):
```

(continues on next page)

(fortsättning från föregående sida)

```
        return QgsServerOgcApi.data

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

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

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

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


class CustomApi():

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

KAPITEL 21

Lathund för PyQGIS

Råd

Kodsnuttarna på den här sidan behöver följande import om du befinner dig utanför pyqgis-konsolen:

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

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

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

21.1 Användargränssnitt

Ändra utseende och känsla

```
from qgis.PyQt.QtWidgets import QApplication

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

Ändra ikon och titel

```
from qgis.PyQt.QtGui import QIcon

```

(continues on next page)

(fortsättning från föregående sida)

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

21.2 Inställningar

Hämta QgsSettings-listan

```
from qgis.core import QgsSettings

qs = QgsSettings()

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

21.3 Verktygsfält

Ta bort verktygsfältet

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

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

Ta bort verktygsfältet för åtgärder

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

21.4 Menyer

Ta bort menyn

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

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

21.5 Canvas

Access canvas

```
canvas = iface.mapCanvas()
```

Ändra färg på duken

```
from qgis.PyQt.QtCore import Qt

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

Intervall för kartuppdatering

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

21.6 Sikt

Lägg till vektorlager

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

Gör aktivt lager

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

Lista alla lager

```
from qgis.core import QgsProject

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

Obtain layers name

```
from qgis.core import QgsVectorLayer
layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer)

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

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

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

I annat fall

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

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

Hitta lager med namn

```
from qgis.core import QgsProject

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

```
layer name you like
```

Ställ in aktivt lager

```
from qgis.core import QgsProject

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

Uppdatera lagret med intervall

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)
# Enable data reloading
layer.setAutoRefreshMode(Qgis.AutoRefreshMode.ReloadData)
```

Visa metoder

```
dir(layer)
```

Lägga till ny funktion med funktionsformulär

```
from qgis.core import QgsFeature, QgsGeometry

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

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

Lägga till ny funktion utan funktionsformulär

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

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

Hämta funktioner

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

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

Få utvalda funktioner

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

Få utvalda funktioner Ids

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

Skapa ett minneslager från utvalda funktioner Ids

```
from qgis.core import QgsFeatureRequest

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

Hämta geometri

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

```
10.000000, 10.000000
```

Flytta geometri

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

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

Ställ in CRS

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

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

Se CRS

```
from qgis.core import QgsProject

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

Dölja en fältkolumn

```
from qgis.core import QgsEditorWidgetSetup

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

(continues on next page)

(fortsättning från föregående sida)

```
        else:
            continue
```

Lager från WKT

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

layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.09 34.89,-88.39 30.34,-89.
↪57 30.18,-89.73 31,-91.63 30.99,-90.87 32.37,-91.23 33.44,-90.93 34.23,-90.30 34.
↪99,-88.82 34.99))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])
```

Ladda alla vektorlager från GeoPackage

```
from qgis.core import QgsDataProvider

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

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

Ladda kakelskikt (XYZ-lager)

```
from qgis.core import QgsRasterLayer, QgsProject

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

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

Avlägsna alla lager

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

Ta bort alla stödlinjer

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

21.7 Innehållsförteckning

Tillgång till kontrollerade lager

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

Ta bort kontextuell meny

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

21.8 Avancerad TOC

Root-nod

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

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

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

node_group.addLayer(layer)

print(root)
print(root.children())
```

Tillgång till den första barnnoden

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

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

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

Hitta grupper och noder

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer

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

(continues on next page)

(fortsättning från föregående sida)

```

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

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

Sök grupp efter namn

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

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

Hitta lager efter id

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

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

Lägg till lager

```
from qgis.core import QgsVectorLayer, QgsProject

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

Lägg till grupp

```
from qgis.core import QgsLayerTreeGroup

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

Flytta laddat lager

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

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

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

parent.removeChildNode(myLayer)
```

Flytta ett laddat lager till en specifik grupp

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

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

Växla synlighet för aktivt lager

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

Är gruppen utvald

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

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

```
False
```

Expandera nod

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

Dold nod trick

```
from qgis.core import QgsProject

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

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

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

Nod signaler

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

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

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

Avlägsna lager

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

Avlägsna grupp

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

Skapa ny innehållsförteckning (TOC)

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

Flytta nod

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

Rename node (namnge nod)

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

21.9 Algoritmer för bearbetning

Hämta algoritmlista

```
from qgis.core import QgsApplication

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

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

Hämta hjälp med algoritmer

Slumpmässig selektion

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

```
...
```

Kör algoritmen

I det här exemplet lagras resultatet i ett tillfälligt minneslager som läggs till i projektet.

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

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

Hur många algoritmer finns det?

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

Hur många leverantörer finns det?

```
from qgis.core import QgsApplication

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

Hur många uttryck finns det?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

21.10 Dekoratörer

Kopieringsrätt

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

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

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

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

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

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

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

```

(continues on next page)

(fortsättning från föregående sida)

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

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

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

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

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

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

21.11 Kompositör

Hämta utskriftslayout med namn

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

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

21.12 Källor

- QGIS Python (PyQGIS) API
- QGIS C++ API
- StackOverFlow QGIS-frågor
- Skript av Klas Karlsson