PyQGIS 3.40 developer cookbook

QGIS Project

03 apr 2025

Indice

CAPITOLO 1

Introduzione

Questo documento vuole essere sia un'esercitazione che un manuale. Anche se non elenca tutti i possibili casi, dovrebbe comunque fornire una buona panoramica delle funzioni principali.

Il supporto Python è stato introdotto per la prima volta in QGIS 0.9. Ci sono diversi modi per utilizzare Python in QGIS Desktop (trattato nelle seguenti sezioni):

- Tramite i comandi nella console Python in QGIS
- Crea e usa plugin
- Esegui automaticamente il codice Python all'avvio di QGIS
- Crea algoritmi di elaborazione
- Crea funzioni per espressioni in QGIS
- Create custom applications based on the QGIS API

Python è disponibile anche per QGIS Server, inclusi i plugin Python (vedi *QGIS Server e Python*) e i collegamenti Python che possono essere utilizzati per incorporare il server QGIS in un'applicazione Python.

C'è un riferimento complete QGIS C++ API che documenta le classi delle librerie QGIS. The Pythonic QGIS API (pyqgis) è quasi identico all'API C++.

Un'altra buona risorsa per imparare come svolgere operazioni comuni è scaricare i plugin esistenti dal repository dei plugin <https://plugins.qgis.org/>`_ ed esaminare il loro codice.

1.1 Scripting nel terminale python

QGIS fornisce un terminale python. Puoi aprirlo dal menu *Plugins* ► *Python Console* :

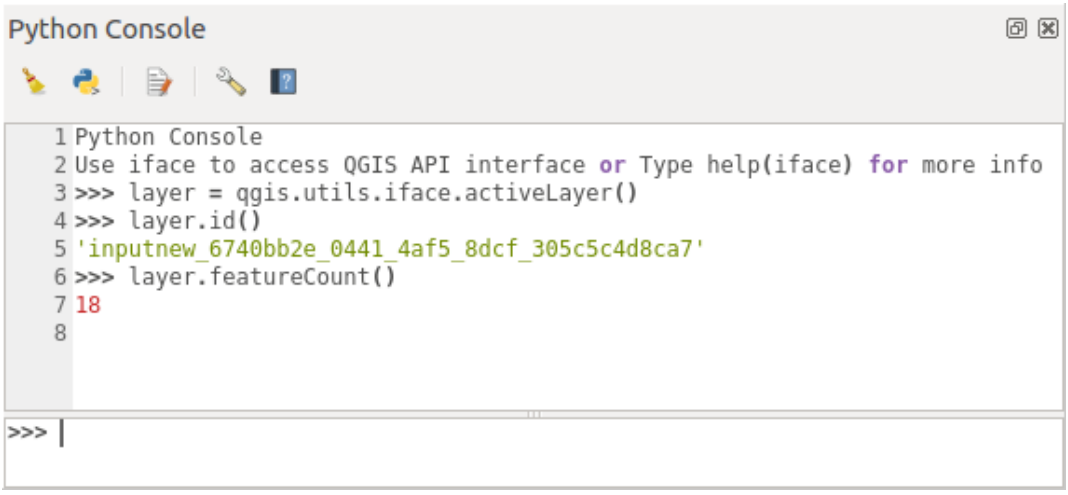


Fig. 1.1: Console python di GIS

La schermata sopra ti mostra come puoi prendere il layer attualmente selezionato nell'elenco dei layer, mostrare il suo ID e, facoltativamente, se è un vettore, ti può mostrare il conteggio degli elementi. Per l'interazione con l'ambiente QGIS, esiste una variabile `iface`, che è un'istanza della **:classe:`QgisInterface <qgis.gui.QgisInterface>`**. Questa interfaccia ti consente l'accesso all'area della mappa, ai menu, alle barre degli strumenti e ad altre parti di QGIS.

Per comodità, esegui le seguenti istruzioni all'avvio del terminale (in futuro sarà possibile impostare ulteriori comandi iniziali)

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

Se fai spesso uso del terminale, può essere utile impostare una scorciatoia per attivarlo (con *Impostazioni* ► *Scorciatoie da tastiera…*)

1.2 Plugin Python

Puoi estendere i comandi di QGIS tramite plugin, che possono essere scritti in python. Il vantaggio principale rispetto ai plugin C ++ è la semplicità della distribuzione (nessuna compilazione per ogni piattaforma) e uno sviluppo più semplice.

Molti plugin che coprono varie funzionalità sono stati scritti dal supporto Python. Il programma di installazione del plugin ti consente di recuperare, aggiornare e rimuovere facilmente i plugin python. Vedi la pagina *Python Plugins <https://plugins.qgis.org/>* per maggiori informazioni sui plugin e sullo sviluppo dei plugin.

Creare un plugin in python è semplice, vedi *Sviluppo di plugin Python* per avere istruzioni dettagliate.

Nota: I plugin python sono disponibili anche per il server QGIS. Per ulteriori dettagli, vedi *QGIS Server e Python* .

1.2.1 Plugin Processing

I Plugin Processing possono essere utilizzati per elaborare i dati. Sono più facili da sviluppare, più specifici e più leggeri dei plugin Python. *Scrivere un plugin di Processing* spiega quando l'uso degli algoritmi di Processing è appropriato e come svilupparli.

1.3 Esegui il codice Python all'avvio di QGIS

Esistono diversi metodi per eseguire il codice Python a ogni avvio di QGIS.

1. Crea uno script startup.py
2. Configura la variabile ambientale `PYQGIS_STARTUP` per un file python esistente
3. Specificando uno script di avvio usando il parametro `--code init_qgis.py`.

1.3.1 Il file `startup.py`

Ad ogni avvio di QGIS, la cartella home di Python dell'utente e un elenco di percorsi di sistema vengono cercati per trovare un file chiamato `startup.py`. Se questo file esiste, viene eseguito dall'interprete Python incorporato.

Il percorso nella cartella home dell'utente si trova solitamente sotto:

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

I percorsi di sistema predefiniti dipendono dal sistema operativo. Per trovare i percorsi che fanno al tuo caso, apri la console di Python ed esegui `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` per vedere l'elenco delle cartelle predefinite.

Lo script `startup.py` viene eseguito immediatamente dopo l'inizializzazione di python in QGIS, nella fase di avvio dell'applicazione.

1.3.2 La variabile ambientale `PYQGIS_STARTUP

Puoi eseguire il codice Python appena prima del completamento dell'inizializzazione di QGIS impostando la variabile ambiente `PYQGIS_STARTUP` sul percorso di un file Python esistente.

Questo codice verrà eseguito prima del completamento dell'inizializzazione di QGIS. Questo metodo è molto utile per pulire sys.path, che può avere percorsi indesiderati, o per isolare/caricare l'ambiente iniziale senza richiedere un ambiente virtuale, ad es. homebrew o MacPorts si installa su Mac.

1.3.3 Il parametro `--code`

Puoi inserire del codice personalizzato da eseguire come parametro di avvio di QGIS. A tale scopo, crea un file python, ad esempio `qgis_init.py`, da eseguire e avviare QGIS dalla riga di comando usando `qgis --code qgis_init.py`.

Il codice fornito tramite ``–code"" viene eseguito in una fase successiva alla fase di inizializzazione di QGIS, dopo che i componenti dell'applicazione sono stati caricati.

1.3.4 Argomenti aggiuntivi per Python

Per fornire argomenti aggiuntivi allo script `--code''` o ad altro codice python che viene eseguito, si può usare l'argomento ``--py-args''. Qualsiasi argomento che viene dopo ``--py-args` e prima di un argomento `--` (se presente) verrà passato a Python ma ignorato dall'applicazione QGIS stessa.

Nell'esempio seguente, `myfile.tif` sarà disponibile tramite `sys.argv` in Python ma non sarà caricato da QGIS. Mentre `otherfile.tif` sarà caricato da QGIS ma non è presente in `sys.argv`.

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

Se vuoi accedere a ogni parametro della linea di comando da Python, puoi usare `QCoreApplication.arguments()`.

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

1.4 Applicazioni Python

È spesso comodo creare degli script per automatizzare i processi. Con PyQGIS, questo è perfettamente possibile — importa il modulo qgis.core, inizializzalo e sei pronto per l'elaborazione.

Oppure potresti voler creare un'applicazione interattiva che usi le funzionalità GIS — eseguire misurazioni, esportare una mappa in PDF, … Il modulo qgis.gui fornisce vari componenti dell'interfaccia grafica, in particolare il widget dell'area grafica della mappa che può essere incorporato nell'applicazione con il supporto per lo zoom, la panoramica e/o qualsiasi altro strumento personalizzato della mappa.

Le applicazioni personalizzate PyQGIS o i singoli script autonomi devono essere configurati per individuare le risorse QGIS, come le informazioni di proiezione e gli operatori per la lettura dei vettori e dei raster. Le risorse di QGIS vengono inizializzate aggiungendo alcune righe all'inizio dell'applicazione o dello script. Il codice per inizializzare QGIS per applicazioni personalizzate e singoli script è simile. Di seguito sono riportati esempi di ciascuno.

Nota: Non usare `qgis.py` come nome per il tuo script. Python non sarà in grado di importare i collegamenti poiché il nome dello script li confonderà.

1.4.1 Usare PyQGIS in script

Per avviare un singolo script, attiva le risorse QGIS all'inizio dello script:

```
from qgis.core import *

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

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

# Load providers
qgs.initQgis()

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

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

Prima importiamo il modulo qgis.core e configuriamo il prefisso del percorso. Il prefisso del percorso è la posizione in cui QGIS è installato sul tuo sistema. Viene configurato nello script chiamando il metodo `setPrefixPath()`. Il secondo parametro di `setPrefixPath()` è impostato a `True`, specificando che i percorsi predefiniti devono essere usati.

Il percorso di installazione di QGIS varia a seconda della piattaforma; il modo più semplice per trovarlo per il tuo sistema è usare il *Scripting nel terminale python* dall'interno di QGIS e controllare il risultato della sua esecuzione:

```
QgsApplication.prefixPath()
```

Dopo aver configurato il prefisso del percorso, salviamo un riferimento a `QgsApplication` nella variabile `qgs`. Il secondo parametro è impostato a `False`, specificando che non abbiamo intenzione di usare la GUI poiché stiamo scrivendo uno script standalone. Con `QgsApplication` configurato, carichiamo i fornitori di dati QGIS e il registro dei layer chiamando il metodo `initQgis()`.

```
qgs.initQgis()
```

Con QGIS inizializzato, siamo pronti a scrivere il resto dello script. Infine, concludiamo chiamando `exitQgis()` per rimuovere i data provider e il registro dei layer dalla memoria.

```
qgs.exitQgis()
```

1.4.2 Usare PyQGIS in applicazioni personalizzate

L'unica differenza tra *Usare PyQGIS in script* e un'applicazione PyQGIS personalizzata è il secondo parametro quando si istanzia la `QgsApplication`. Imposta `True` invece di `False` per indicare che abbiamo intenzione di usare una GUI.

```
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()
```

Ora puoi lavorare con l'API QGIS: caricare i livelli ed eseguire alcune elaborazioni o avviare una GUI con un'area della mappa. Le possibilità sono infinite :-)

1.4.3 Avviare applicazioni personalizzate

Devi dire al tuo sistema dove cercare le librerie QGIS e i moduli Python se non si trovano in una posizione già nota, altrimenti Python lo segnalerà:

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

Puoi risolvere impostando la variabile ambiente `PYTHONPATH`. Nei seguenti comandi, `<qgispath>` deve essere sostituito con il percorso di installazione di QGIS:

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

Ora, il percorso dei moduli PyQGIS è noto, ma questi dipendono dalle librerie `qgis_core` e `qgis_gui` (i moduli Python servono solo come contenitori). Il percorso di queste librerie potrebbe essere sconosciuto al sistema operativo, quindi verrà visualizzato nuovamente un errore di importazione (il messaggio potrebbe variare in base al sistema):

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

Risolvi questo aggiungendo le cartelle delle librerie QGIS nel percorso di ricerca del collegamento dinamico:

- su Linux: **`export LD_LIBRARY_PATH=/<qgispath>/lib`**
- su Windows: **`set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%`** dove `<qgisrelease>` dovrebbe essere sostituito con il tuo tipo di versione (es. `qgis-ltr`, `qgis`, `qgis-dev`)

Questi comandi possono essere inseriti in uno script che se ne occuperà all'avvio del programma. Quando si fa uso di applicazioni personalizzate utilizzando PyQGIS, esistono di solito due possibilità:

- richiede all'utente di installare QGIS prima di installare l'applicazione. Il programma di installazione dell'applicazione dovrebbe cercare percorsi predefiniti delle librerie QGIS e consentire all'utente di impostare il percorso se non trovato. Questo approccio ha il vantaggio di essere più semplice, tuttavia richiede all'utente di fare più passaggi.
- impacchetta QGIS insieme alla tua applicazione. Rilasciare l'applicazione può essere più impegnativo e il pacchetto sarà più grande, ma l'utente sarà risparmiato dall'onere di scaricare e installare parti addizionali del programma.

I due modelli di distribuzione possono essere combinati. È possibile fornire applicazioni indipendenti su Windows e macOS, ma per Linux lasciare l'installazione di GIS all'utente e al suo gestore dei pacchetti.

1.5 Note tecniche su PyQt e SIP

Python è una dei linguaggi preferiti per lo scripting. I collegamenti PyQGIS in QGIS 3 dipendono da SIP e PyQt5. La ragione per usare SIP invece del SWIG più usato è che il codice QGIS dipende dalle librerie Qt. I collegamenti Python per Qt (PyQt) vengono eseguiti utilizzando SIP e ciò consente una perfetta integrazione di PyQGIS con PyQt.

CAPITOLO 2

Caricamento Progetti

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

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

Ti può capitare di dovere caricare un progetto esistente da un plugin o (più spesso) quando stai sviluppando un'applicazione QGIS Python autonoma (see: *Applicazioni Python*).

Per caricare un progetto nell'applicazione QGIS corrente devi creare un'istanza della classe `QgsProject`. Questa è una classe singleton, quindi devi utilizzare il suo metodo `instance()` per farlo. Puoi chiamare il suo metodo `read()`, indicando il percorso del progetto da caricare:

```
# 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
```

Se vuoi modificare il progetto (ad esempio per aggiungere o rimuovere dei layer) e salvare le modifiche, chiama il metodo `write()` dell'istanza del progetto. Il metodo `write()` accetta anche un nuovo percorso per salvare il progetto in una nuova posizione:

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

Sia la funzione `read()` che quella `write()` restituiscono un valore booleano che puoi utilizzare per verificare se l'operazione ha avuto esito positivo.

Nota: Se stai scrivendo un'applicazione QGIS, per sincronizzare il progetto caricato con l'area di mappa, devi creare un'istanza di `QgsLayerTreeMapCanvasBridge` come nell'esempio seguente:

```
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 Risoluzione di percorsi errati

Può accadere che i layer caricati nel progetto vengano spostati in un'altra posizione. Quando il progetto viene caricato di nuovo, tutti i percorsi dei layer sono interrotti. La classe `QgsPathResolver` aiuta a riscrivere il percorso dei layer all'interno del progetto.

Il suo metodo `setPathPreprocessor()` permette di impostare una funzione di pre-processore del percorso personalizzata per manipolare i percorsi e le fonti di dati prima di risolverli in riferimenti di file o origini layer.

La funzione di elaborazione deve accettare un singolo argomento stringa (che rappresenta il percorso del file originale o l'origine dei dati) e restituire una versione elaborata di questo percorso. La funzione di preprocessore del percorso viene chiamata **prima** di qualsiasi gestore di layer errato. Se sono impostati più preprocessori, essi verranno richiamati in sequenza, in base all'ordine in cui sono stati originariamente impostati.

Alcuni casi d'uso:

1. sostituire un percorso obsoleto:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

2. sostituire un indirizzo host del database con uno nuovo:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. sostituire le credenziali del database memorizzate con quelle nuove:

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

Allo stesso modo, è disponibile un metodo `setPathWriter()` per una funzione di scrittura del percorso.

Un esempio per sostituire il percorso con una variabile:

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

QgsPathResolver.setPathWriter(my_processor)
```

Entrambi i metodi restituiscono un ``id"" che può essere usato per rimuovere il preprocessore o lo scritto che hanno aggiunto. Vedere `removePathPreprocessor()` e `removePathWriter()`.

2.2 Usare dei flag per velocizzare le cose

In alcuni casi in cui non è necessario utilizzare un progetto completamente funzionante, ma si desidera accedervi solo per un motivo specifico, i flag possono essere utili. Un elenco completo di flag è disponibile sotto `ProjectReadFlag`. È possibile cumulare più flag.

Ad esempio, se non ci interessano i layer e i dati effettivi e vogliamo semplicemente accedere a un progetto (ad esempio per il layout o le impostazioni della vista 3D), possiamo usare il flag `DontResolveLayers` per bypassare la fase di convalida dei dati e impedire che appaia la finestra di dialogo dei layer errati. Si può fare come segue:

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

Per aggiungere altri flag è necessario utilizzare l'operatore OR Bitwise di python (`|`).

CAPITOLO 3

Caricamento Layer

Suggerimento: I frammenti di codice in questa pagina hanno bisogno dei seguenti import:

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

Apri alcuni layer con dati. QGIS riconosce vettori e raster. inoltre sono disponibili tipi di layer personalizzati ma non li discuteremo qui.

3.1 Vettori

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

Per un vettore geopackage:

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

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

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

Questo crea un nuovo layer e lo aggiunge al progetto QGIS corrente (facendolo apparire nell'elenco dei layer) in un solo passaggio. La funzione restituisce l'istanza del layer o `None` se non è stato possibile caricare il layer.

Il seguente elenco mostra come accedere a varie fonti di dati usando i fornitori di dati vettoriali:

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

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

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

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

  - per dxf (nota le opzioni nell'origine dati uri):

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

- Database PostGIS - data source è una stringa con tutte le informazioni necessarie per creare una connessione al database PostgreSQL.

  La classe `QgsDataSourceUri` può generare questa stringa per te. Nota che QGIS deve essere stato compilato con il supporto Postgres, altrimenti questo provider non è disponibile:

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

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

Nota: L'argomento `False` passato a `uri.uri(False)` previene l'espansione dei parametri di configurazione di autenticazione, se tu non usi nessuna configurazione di autenticazione questo argomento non produrrà nessun effetto.

- CSV o altri file di testo delimitati — per aprire un file con un punto e virgola come delimitatore, con il campo «x» per la coordinata X e il campo «y» per la coordinata Y si userebbe qualcosa di simile:

```
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)
```

Nota: La stringa provider è strutturata come un URL, quindi il percorso deve avere il prefisso `file://`. Inoltre permette di formattare geometrie WKT (well known text) in alternativa ai campi `x`` e `y`, e permette di specificare il sistema di riferimento di coordinate. Per esempio:

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

- I file GPX – il provider di dati «gpx» legge tracce, percorsi e waypoint dai file gpx. Per aprire un file, il tipo (track/route/waypoint) deve essere specificato come parte dell'url:

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

- Database SpatiaLite — Analogamente ai database PostGIS, :class:`Qgsdatasourceuri <qgis.core.Qgsdatasourceuri>`può essere utilizzato per la generazione dell'identificatore della fonte di dati:

```
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)
```

- Geometrie basate su MySQL WKB, tramite GDAL — l'origine dei dati è la stringa di connessione alla tabella:

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

- Connessione WFS: la connessione è definita con un URI e usando il ``WFS``provider:

```
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")
```

L'uri può essere creato usando la libreria standard `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))
```

Nota: Puoi modificare l'origine dei dati di un layer esistente chiamando `setDataSource()` su un'istanza `QgsVectorLayer`, come nell'esempio seguente:

```
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 Raster

Per accedere ai file raster, viene utilizzata la libreria GDAL. Supporta una vasta gamma di formati di file. In caso di problemi con l'apertura di alcuni file, verificare se il GDAL ha il supporto per il formato particolare (non tutti i formati sono disponibili per impostazione predefinita). Per caricare un raster da un file, specificare il nome del file e il nome di visualizzazione:

```
# 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!")
```

Per caricare un raster da un geopackage:

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

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

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

Analogamente ai livelli vettoriali, i livelli raster possono essere caricati usando la funzione addRasterLayer dell'oggetto classe:*QgisInterface <qgis.gui.QgisInterface>*:

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

Questo crea un nuovo livello e lo aggiunge al progetto corrente (facendolo apparire nella lista dei livelli) in un solo passaggio.

Per caricare un raster Postgis:

I raster PostGIS, simili ai vettori PostGIS, possono essere aggiunti a un progetto usando una stringa URI. È efficiente mantenere un dizionario riutilizzabile di stringhe per i parametri di connessione al database. Questo rende facile modificare il dizionario per la connessione appropriata. Il dizionario viene poi codificato in un URI usando l'oggetto metadati del provider "postgraster". Dopo di che il raster può essere aggiunto al progetto.

```
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
```

(continues on next page)

(continua dalla pagina precedente)

```
    '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
↪")
```

I livelli raster possono anche essere creati da un servizio WCS:

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

Ecco una descrizione dei parametri che l'URI WCS può contenere:

L'URI WCS è composto da coppie **key=value** separate da `&`. È lo stesso formato come la stringa di interrogazione nell'URL, codificata nello stesso modo. : class:*Qgsdatasourceuri qgis.core.Qgsdatasourceuri*> dovrebbe essere usato per costruire l'URI per garantire che i caratteri speciali siano codificati correttamente.

- **url** (obbligatorio) : URL del server WCS. Non usare VERSION nell" URL, perché ogni versione del WCS usa un nome di parametro diverso per la versione **Getcapabilities**, vedi param version.
- **identifier** (obbligatorio) : Coverage name
- **time** (opzionale) : posizione temporale o periodo temporale (beginPosition/endposition[/timeResolution])
- **format** (opzionale) : Nome formato supportato. Il formato predefinito è il primo formato supportato con tif nel nome o il primo formato supportato.
- **crs** (facoltativo): CRS nel modulo AUTHORITY:ID, ad es. EPSG:4326. Il valore predefinito è EPSG:4326 se supportato o il primo SR supportato.
- **username** (opzionale) : Nome utente per l'autenticazione di base.
- **password** (opzionale) : password per l'autenticazione di base.
- **IgnoreGetMapUrl** (opzionale, hack) : Se specificato (impostato a 1), ignora l'URL di GetCoverage pubblicizzato da GetCapabilities. Può essere necessario se un server non è configurato correttamente.
- **InvertAxisOrientation** (opzionale, hack): Se specificato (impostato a 1), inverte l'asse nella richiesta Getcoverage. Può essere necessario per un SR geografico se un server sta" usando un'ordine errato dell'asse.
- **IgnoreAxisOrientation** (opzionale, hack): Se specificato (impostato a 1), non invertire l'orientamento dell'asse secondo lo standard WCS per SR geografico.

- **cache** (opzionale) : controllo del carico della cache, come descritto in QNetworkRequest::CacheLoadControl, ma la richiesta viene rinviata come PreferCache se non riuscita con AlwaysCcache. Valori ammessi: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. Il valore predefinito è AlwaysCache.

In alternativa è possibile caricare un livello raster dal server WMS. Tuttavia attualmente non è possibile accedere alla risposta GetCapabilities da API — devi sapere quali livelli vuoi:

```
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 Istanza QgsProject

Se volete usare i livelli aperti per il rendering, non dimenticate di aggiungerli all'istanza `QgsProject qgis.core.QgsProject>`. L'istanza `QgsProject` prende la proprietà di livelli e possono essere successivamente accessibili da qualsiasi parte dell'applicazione dal loro ID univoco. Quando il livello viene rimosso dal progetto, viene anche eliminato. I livelli possono essere rimossi dall'utente nell'interfaccia QGIS, o tramite Python usando il metodo `removeMapLayer() qgis.core.Qgsproject.removeMapLayer>()`.

L'aggiunta di un livello al progetto corrente viene effettuata utilizzando il metodo `addMapLayer()` :

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

Per aggiungere un livello in una posizione assoluta:

```
# 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))
```

Se si desidera eliminare il livello utilizzare il metodo `removeMapLayer()` :

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

Nel codice precedente, l'id del livello viene passato (si può ottenere chiamando il metodo `id()` del livello), ma si può anche passare l'oggetto del livello stesso.

Per una lista di livelli caricati e id di livello, usare il metodo `mapLayers()` :

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

CAPITOLO 4

Accedere alla Legenda (o Table Of Contents - TOC)

Suggerimento: I frammenti di codice di questa pagina necessitano delle seguenti importazioni se sei è al di fuori della console pyqgis:

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

Puoi usare diverse classi per accedere a tutti i layer caricati nella legenda e usarli per recuperare informazioni:

- `QgsProject`
- `QgsLayerTreeGroup`

4.1 La classe QgsProject

Puoi usare `QgsProject` per recuperare informazioni sulla legenda e su tutti i layer caricati.

Devi creare un"`instanza` di `QgsProject` e utilizzare i suoi metodi per ottenere i layer caricati.

Il metodo principale è `mapLayers()`. Restituisce un dizionario dei layer caricati:

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

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

Le ``chiavi"" del dizionario sono gli id univoci dei layer, mentre i ``valori"" sono gli oggetti correlati.

A questo punto è facile ottenere qualsiasi altra informazione sui layer:

```
# 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
layers_list = {}
```

(continues on next page)

(continua dalla pagina precedente)

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

print(layers_list)
```

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

Puoi anche interrogare la legenda utilizzando il nome del layer:

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

Nota: Viene restituito un elenco con tutti i layer corrispondenti, quindi si indicizza con `[0]` per ottenere il primo layer con questo nome.

4.2 Classe QgsLayerTreeGroup

L'albero dei layer è una classica struttura ad albero composta da nodi. Attualmente esistono due tipi di nodi: nodi di gruppo (`QgsLayerTreeGroup`) e nodi di layer (`QgsLayerTreeLayer`).

Nota: for more information you can read these blog posts of Martin Dobias: Part 1 Part 2 Part 3

L'albero dei layer del progetto è facilmente accessibile con il metodo `layerTreeRoot()` della classe `QgsProject`:

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

`root` è un nodo di gruppo e ha *children*:

```
root.children()
```

Viene restituito un elenco di figli diretti. I figli del sottogruppo devono essere accessibili dal proprio genitore diretto.

Possiamo recuperare uno dei figli:

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

```
<QgsLayerTreeLayer: countries>
```

I layer possono anche essere recuperati usando il loro ``id"" (univoco):

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

E anche I gruppi possono essere cercati utilizzando i loro nomi:

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

`QgsLayerTreeGroup` ha molti altri metodi utili che possono essere usati per ottenere ulteriori informazioni sulla TOC:

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

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

Ora aggiungiamo alcuni layer all'albero dei layer del progetto. Ci sono due modi per farlo:

1. **Explicit addition** usando il `addLayer()` o `insertLayer()` functions:

```
# 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. **Aggiunta implicita**: poiché l'albero dei layer del progetto è collegato al registro dei layer, è sufficiente aggiungere un layer al registro dei layer della mappa:

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

Puoi passare facilmente da `QgsVectorLayer` a `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)>
```

I gruppi possono essere aggiunti con il metodo `addGroup()`. Nell'esempio che segue, il primo aggiungerà un gruppo alla fine della legenda, mentre il secondo può aggiungere un altro gruppo all'interno di uno esistente:

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

Per spostare nodi e gruppi esistono molti metodi validi.

Lo spostamento di un nodo esistente avviene in tre passi:

1. clonazione del nodo esistente
2. spostamento del nodo clonato nella posizione desiderata
3. eliminazione del nodo originale

```
# 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)
```

È un po" più *complicato* spostare un layer nella legenda:

```
# 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()
# 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)
```

(continues on next page)

(continua dalla pagina precedente)

```
# remove the original myvl
root.removeChildNode(myvl)
```

o spostarlo in un gruppo esistente:

```
# 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)
```

Alcuni altri metodi che possono essere utilizzati per modificare gruppi e layer:

```
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)
```

CAPITOLO 5

Usare Layer Raster

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

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

5.1 Dettagli del raster

Un layer raster è costituito da una o più bande raster, denominate raster a banda singola e raster multibanda. Una banda rappresenta una matrice di valori. Un'immagine a colori (ad esempio una foto aerea) è un raster composto da bande rosse, blu e verdi. I raster a banda singola rappresentano tipicamente variabili continue (ad esempio, l'altitudine) o discrete (ad esempio, l'uso del suolo). In alcuni casi, un layer raster viene fornito con una tavolozza e i valori raster si riferiscono ai colori memorizzati nella tavolozza.

Il codice seguente presuppone che `rlayer` sia un oggetto `QgsRasterLayer`.

```
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 Visualizzatore

Quando un layer raster viene caricato, prende una visualizzazione predefinita in base al suo tipo. Può essere modificato nelle proprietà del layer o a programma.

Per consultare il visualizzatore corrente:

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

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

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

```
singlebandgray
```

Per impostare una visualizzazione, utilizza il metodo `setRenderer()` di `QgsRasterLayer`. Esistono diverse classi di visualizzatori (derivate da `QgsRasterRenderer`):

- `QgsHillshadeRenderer`
- `QgsMultiBandColorRenderer`

- QgsPalettedRasterRenderer
- QgsRasterContourRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

I layer raster a banda singola possono essere disegnati con colori grigi (valori bassi = nero, valori alti = bianco) o con un algoritmo pseudocolore che assegna i colori ai valori. Anche i raster a banda singola con tavolozza possono essere disegnati utilizzando la tavolozza. I layer multibanda sono in genere disegnati mappando le bande ai colori RGB. Un'altra possibilità è quella di utilizzare una sola banda per il disegno.

5.2.1 Raster a Banda Singola

Supponiamo di voler visualizzare un layer raster a banda singola con colori che vanno dal verde al giallo (corrispondenti a valori di pixel da 0 a 255). Nella prima fase prepareremo un oggetto QgsRasterShader e configureremo la sua funzione shader:

```
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)
```

Lo shader mappa i colori come specificato dalla sua mappa dei colori. La mappa dei colori viene fornita come un elenco di valori di pixel con i colori associati. Esistono tre modalità di interpolazione:

- lineare (Interpolated): il colore è interpolato linearmente dalle registrazioni della mappa di colore sopra e sotto il valore del pixel
- discreto (Discrete): il colore viene preso dalla registrazione della mappa di colore più vicina con valore uguale o maggiore
- exact (Exact): il colore non viene interpolato, verranno disegnati solo i pixel con valori uguali alle registrazioni della mappa di colore.

Nel secondo passo assoceremo questo shader al layer raster:

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

Il numero 1 nel codice precedente è il numero di banda (le bande raster sono indicizzate a partire da uno).

Infine dobbiamo usare il metodo triggerRepaint() per vedere i risultati:

```
rlayer.triggerRepaint()
```

5.2.2 Raster Multi Banda

Per impostazione predefinita, QGIS mappa le prime tre bande in rosso, verde e blu per creare un'immagine a colori (questo è lo stile di disegno `MultiBandColor`). In alcuni casi puoi sovrascrivere queste impostazioni. Il codice seguente scambia la banda rossa (1) con quella verde (2):

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

Nel caso in cui sia necessaria una sola banda per la visualizzazione del raster, puoi scegliere un disegno a banda singola, con livelli di grigio o pseudocolore.

Dobbiamo usare `triggerRepaint()` per aggiornare la mappa e vedere il risultato:

```
rlayer_multi.triggerRepaint()
```

5.3 Valori dell'interrogazione

I valori raster possono essere interrogati utilizzando il metodo `sample()` della classe `QgsRasterDataProvider`. Devi specificare un `QgsPointXY` e il numero di banda del layer raster che vuoi interrogare. Il metodo restituisce una tupla con il valore e `True` o `False` a seconda dei risultati:

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

Un altro metodo per interrogare i valori raster è il metodo `identify()` che restituisce un oggetto `QgsRasterIdentifyResult`.

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

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

```
{1: 323.0}
```

In questo caso, il metodo `results()` restituisce un dizionario, con gli indici di banda come chiavi e i valori di banda come valori. Per esempio, qualcosa come `{1: 323.0}`

5.4 Modificare dati raster

Puoi creare un layer raster usando la classe `QgsRasterBlock`. Ad esempio, per creare un blocco raster 2x2 con un byte per pixel:

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

I pixel raster possono essere sovrascritti grazie al metodo `writeBlock()`. Per sovrascrivere i dati raster esistenti nella posizione 0,0 con un blocco 2x2:

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

CAPITOLO 6

Usare Layer Vettoriali

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

```
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,
)
```

Questa sezione riassume le varie azioni che si possono eseguire con i vettori.

La maggior parte del materiale presente in questo documento si basa sui metodi della classe `QgsVectorLayer`.

6.1 Recuperare informazioni sugli attributi

Puoi recuperare informazioni sui campi associati a un layer vettoriale chiamando `fields()` su un oggetto `QgsVectorLayer`:

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

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

I metodi `displayField()` e `mapTipTemplate()` forniscono informazioni sul campo e sul modello utilizzati nella scheda maptips.

Quando carichi un layer vettoriale, un campo viene sempre scelto da QGIS come `Display Name`, mentre `Suggerimenti Mappa in HTML` è vuoto per default. Con questi metodi puoi facilmente ottenere entrambi:

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

```
name
```

Nota: Se cambi il `Display Name` da un campo a un'espressione, devi usare `displayExpression()` invece di `displayField()`.

6.2 Iterare un Vettore.

L'iterazione sugli elementi di un layer vettoriale è una delle operazioni più comuni. Di seguito è riportato un esempio di semplice codice di base per eseguire questo processo e per mostrare alcune informazioni su ciascun elemento. Si presume che la variabile `layer` abbia un oggetto `QgsVectorLayer`.

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

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

(continues on next page)

(continua dalla pagina precedente)

```
    # 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 Selezionare elementi

In QGIS desktop, gli elementi possono essere selezionati in diversi modi: l'utente può fare clic su un elemento, disegnare un rettangolo sull" area della mappa o utilizzare un filtro di espressione. Gli elementi selezionati sono normalmente evidenziati con un colore diverso (il colore predefinito è il giallo) per attirare l'attenzione dell'utente sulla selezione.

A volte può essere utile selezionare automaticamente gli elementi o cambiare il colore predefinito.

Per selezionare tutti gli elementi, può essere usato il metodo `selectAll()`:

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

Per effettuare una selezione usando un'espressione, usa il metodo `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)
layer = iface.activeLayer()
```

(continues on next page)

(continua dalla pagina precedente)

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

Per cambiare il colore delle selezioni puoi usare il metodo `setSelectionColor()` di `QgsMapCanvas` come mostrato nel seguente esempio:

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

Per aggiungere gli elementi all'elenco degli elementi selezionati per un dato layer, puoi chiamare `select()` passandogli l'elenco degli ID degli elementi:

```
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)
```

Per annullare la selezione:

```
layer.removeSelection()
```

6.3.1 Accedere agli attributi

Gli attributi possono essere indicati con il loro nome:

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

```
First feature
```

In alternativa, si può fare riferimento agli attributi tramite l'indice. Questo è un po" più veloce rispetto all'uso del nome. Per esempio, per ottenere il secondo attributo:

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

```
First feature
```

6.3.2 Iterare gli elementi selezionati

Se ti servono solo gli elementi selezionati, puoi usare il metodo `selectedFeatures()` per il layer vettoriale:

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

6.3.3 Iterare un sottoinsieme di caratteristiche

Se vuoi iterare su un determinato sottoinsieme di elementi in un layer, come quelli all'interno di una determinata area, devi aggiungere un oggetto `QgsFeatureRequest` alla chiamata `getFeatures()`. Ecco un esempio:

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

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

Per motivi di velocità, l'intersezione viene spesso eseguita solo utilizzando il rettangolo di selezione degli elementi. Esiste tuttavia un flag `ExactIntersect` che assicura che vengano restituite solo gli elementi che si intersecano:

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

Con `setLimit()` puoi limitare il numero di elementi che vengono richiesti. Ecco un esempio:

```
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>
```

Se hai bisogno di un filtro basato su attributi, invece (o in aggiunta) di uno spaziale come mostrato negli esempi precedenti, puoi costruire un oggetto `QgsExpression` e passarlo al costruttore `QgsFeatureRequest`. Ecco un esempio:

```
# 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)
```

Vedi *Espressioni, Filtraggio e Calcolo di Valori* per i dettagli sulla sintassi supportata da `QgsExpression`.

La richiesta può essere usata per definire i dati recuperati per ogni elemento, in modo che l'iteratore restituisca tutti gli elementi, restituendo però dati parziali per ciascuno di essi.

```
# 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 Modificare i Vettori

La maggior parte dei fornitori di dati vettoriali supporta la modifica dei dati dei layer. A volte supportano solo un sottoinsieme di possibili azioni di modifica. Usa la funzione `capabilities()` per scoprire quale insieme di funzionalità è supportato.

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

```
The layer supports DeleteFeatures
```

Per un elenco di tutte le capabilities disponibili, consulta la `Documentazione API di QgsVectorDataProvider`.

Per stampare la descrizione testuale delle capabilities del layer in un elenco separato da virgole, puoi usare `capabilitiesString()` come nell'esempio seguente:

```
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'
```

Utilizzando uno dei seguenti metodi per la modifica dei layer vettoriali, le modifiche vengono direttamente apportate all'archivio dati sottostante (un file, un database, ecc.). Se vuoi apportare solo modifiche temporanee, passa alla sezione successiva che spiega come effettuare *modifications with editing buffer*.

Nota: Se stai lavorando all'interno di QGIS (sia dalla console che da un plugin), potrebbe essere necessario forzare un ridisegno della mappa per vedere le modifiche che hai fatto alla geometria, allo stile o agli attributi:

```
# 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 Aggiungi Elementi

Creare alcune istanze `QgsFeature` e passarne un elenco al metodo `QgsVectorDataProvider` `addFeatures()` del provider. Esso restituirà due valori: il risultato (`True` o `False`) e l'elenco degli elementi aggiunti (il loro ID è impostato dal data store).

Per impostare gli attributi dell" elemento, si può inizializzare l'elemento passando un oggetto `QgsFields` (si può ottenere dal metodo `fields()` del layer vettoriale) o chiamare `initAttributes()` passando il numero di campi che vuoi aggiungere.

```
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 Eliminare Elementi

Per eliminare alcuni elementi, è sufficiente fornire un elenco degli ID degli elementi stessi.

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

6.4.3 Modificare Elementi

È possibile modificare la geometria dell" elemento o modificare alcuni attributi. L'esempio seguente modifica prima i valori degli attributi con indice 0 e 1 e poi la geometria dell" elemento.

```
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 })
```

Suggerimento: Preferibile la classe QgsVectorLayerEditUtils per modifiche solo geometriche

Se hai solo bisogno di modificare le geometrie, potresti considerare di usare la `QgsVectorLayerEditUtils` che fornisce alcuni metodi utili per modificare le geometrie (traslazione, inserimento o spostamento di vertici, ecc.).

6.4.4 Modificare i Vettori con un Buffer di Modifica

Quando modifichi i vettori all'interno dell'applicazione QGIS, devi prima avviare la modalità di modifica per un particolare layer, poi apportare alcune modifiche e infine eseguire il commit (o il rollback) delle modifiche. Tutte le modifiche apportate non vengono scritte finché non esegui il commit, ma rimangono nel buffer di modifica in memoria del layer. È possibile usare questa funzionalità anche in modo programmatico: è solo un altro metodo per la modifica dei layer vettoriali che integra l'uso diretto dei fornitori di dati. Utilizzare questa opzione quando si forniscono strumenti dell'interfaccia grafica per la modifica dei layer vettoriali, in quanto consente all'utente di decidere se eseguire il commit/rollback e permette l'uso di annullo/ripristina. Quando le modifiche vengono impegnate, tutte le modifiche del buffer di modifica vengono salvate nel fornitore di dati.

I metodi sono simili a quelli visti nel provider, ma vengono richiamati sull'oggetto `QgsVectorLayer`.

Affinché questi metodi funzionino, il layer deve essere in modalità di modifica. Per avviare la modalità di modifica, utilizza il metodo `startEditing()`. Per interrompere la modifica, utilizza i metodi `commitChanges()` o `rollBack()`. Il primo impegna tutte le modifiche apportate all'origine dati, mentre il secondo le scarta e non modifica affatto l'origine dati.

Per scoprire se un layer è in stato di modifica, usa il metodo `isEditable()`.

Di seguito hai alcuni esempi che dimostrano come utilizzare questi metodi di editing.

```
from qgis.PyQt.QtCore import QMetaType

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

# add two features (QgsFeature instances)
layer.addFeatures([feat1,feat2])
```

(continues on next page)

(continua dalla pagina precedente)

```
# 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)
```

Per far funzionare correttamente annulla/ripristina, le chiamate di cui sopra devono essere racchiuse in comandi di annullamento. (Se non ci si preoccupa dell'annulla/ripristina e si vuole che le modifiche vengano memorizzate immediatamente, si potrà lavorare più facilmente con *editing with data provider*).

Ecco come utilizzare la funzionalità di annullamento:

```
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()
```

Il metodo `beginEditCommand()` crea un comando interno «attivo» e registra le modifiche successive nel layer vettoriale. Con la chiamata a `endEditCommand()` il comando viene spinto nella lista di annullamento e l'utente potrà annullarlo/ripristinarlo dalla GUI. Nel caso in cui qualcosa sia andato storto durante le modifiche, il metodo `destroyEditCommand()` rimuoverà il comando e annullerà tutte le modifiche effettuate mentre il comando era attivo.

Puoi anche usare la dichiarazione `with edit(layer)` per racchiudere commit e rollback in un blocco di codice più semantico, come mostrato nell'esempio seguente:

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

Questo chiamerà automaticamente `commitChanges()` alla fine. Se si verifica un'eccezione, `rollBack()` tutte le modifiche. Nel caso in cui si verifichi un problema all'interno di `commitChanges()` (quando il metodo restituisce False) verrà sollevata un'eccezione `QgsEditError`.

6.4.5 Aggiungere e Rimuovere Campi

Per aggiungere campi (attributi), è necessario specificare un elenco di definizioni di campi. Per eliminare i campi è sufficiente fornire un elenco di indici di campo.

```
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)
```

Dopo aver aggiunto o rimosso campi nel fornitore di dati, è necessario aggiornare i campi del layer, perché le modifiche non vengono propagate automaticamente.

```
layer.updateFields()
```

Suggerimento: Salva direttamente le modifiche usando `con` il comando basato su

Utilizzando `with edit(layer):` le modifiche saranno apportate automaticamente chiamando `commitChanges()` alla fine. Se si verifica un'eccezione, si attiva `rollBack()` per tutte le modifiche. Vedi *Modificare i Vettori con un Buffer di Modifica*.

6.5 Usare l'Indice Spaziale

Gli indici spaziali possono migliorare notevolmente le prestazioni del codice se è necessario eseguire interrogazioni frequenti su un layer vettoriale. Immaginiamo, ad esempio, di scrivere un algoritmo di interpolazione e che per una determinata posizione sia necessario conoscere i 10 punti più vicini di un layer di punti, per poterli utilizzare per calcolare il valore interpolato. Senza un indice spaziale, l'unico modo che QGIS ha per trovare questi 10 punti è calcolare la distanza di ogni singolo punto dalla posizione specificata e poi confrontare le distanze. Questa operazione può richiedere molto tempo, soprattutto se deve essere ripetuta per diverse posizioni. Se esiste un indice spaziale per il layer, l'operazione è molto più efficace.

Pensa a un layer senza indice spaziale come a un elenco telefonico in cui i numeri di telefono non sono ordinati o indicizzati. L'unico modo per trovare il numero di telefono di una determinata persona è leggere dall'inizio fino a trovarlo.

Gli indici spaziali non vengono creati di default per un layer vettoriale di QGIS, ma puoi crearli facilmente. Ecco cosa devi fare:

- crea un indice spaziale utilizzando la classe `QgsSpatialIndex`:

```
index = QgsSpatialIndex()
```

- aggiungi elementi all'indice — L'indice prende l'oggetto `QgsFeature` e lo aggiunge alla struttura dati interna. Puoi creare l'oggetto manualmente o utilizzarne uno da una precedente chiamata al metodo `getFeatures()` del provider.

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

- In alternativa, puoi caricare tutti gli elementi di un layer in una volta sola, utilizzando il caricamento in blocco

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

- una volta che l'indice spaziale è stato riempito con alcuni valori, puoi eseguire delle query

```
# 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))
```

Puoi utilizzare anche l'indice spaziale `QgsSpatialIndexKDBush`. Questo indice è simile a quello *standard* `QgsSpatialIndex` ma:

- supporta **solo** elementi a punto singolo
- è **statico** (nessuna elemento aggiuntivo può essere aggiunto all'indice dopo la costruzione)
- è **molto più veloce!**
- permette di recuperare direttamente i punti degli elementi di partenza, senza richiedere ulteriori richieste di elementi
- supporta le ricerche *basate sulla distanza*, cioè restituisce tutti i punti entro un raggio da un punto di ricerca.

6.6 La classe QgsVectorLayerUtils

La classe `QgsVectorLayerUtils` contiene alcuni metodi molto utili che si possono usare con i layer vettoriali.

Ad esempio, il metodo `createFeature()` prepara una `QgsFeature` da aggiungere a un layer vettoriale, mantenendo tutti gli eventuali vincoli e valori predefiniti di ogni campo:

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

Il metodo `getValues()` consente di ottenere rapidamente i valori di un campo o di un'espressione:

```
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 Creare Layer vettoriali

Ci sono diversi modi per generare un dataset di layer vettoriali:

- the `QgsVectorFileWriter` class: A convenient class for writing vector files to disk, using either a static call to `writeAsVectorFormatV3()` which saves the whole vector layer or creating an instance of the class and issue calls to inherited `addFeature()`. This class supports all the vector formats that GDAL supports (GeoPackage, Shapefile, GeoJSON, KML and others).
- la classe `QgsVectorLayer`: istanzia un fornitore di dati che interpreta il percorso fornito (url) dell'origine dati per connettersi e accedere ai dati. Può essere usata per creare layer temporanei, basati sulla memoria (`memory`) e connettersi a insiemi di dati vettoriali GDAL (`ogr`), database (`postgres`, `spatialite`, `mysql`, `mssql`) e altro (`wfs`, `gpx`, `delimitedtext`...).

6.7.1 Da un'istanza di `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)
```

Puoi convertire i campi per renderli compatibili con formati diversi, utilizzando la `FieldValueConverter`. Ad esempio, per convertire i tipi di variabili array (ad esempio in Postgres) in un tipo di testo, puoi procedere come segue:

```
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
```

Si può anche specificare un SR di destinazione — se un'istanza valida di `QgsCoordinateReferenceSystem` viene passata come quarto parametro, il layer viene trasformato in tale SR.

Per i nomi dei driver validi, chiama il metodo `supportedFiltersAndFormats()` o consulta i formati supportati da OGR — devi passare il valore nella colonna «Code» come nome del driver.

Opzionalmente puoi impostare se esportare solo gli elementi selezionati, se passare altre opzioni specifiche del driver per la creazione o se dire allo scrittore di non creare attributi… Esiste una serie di altri parametri (opzionali); vedere la documentazione di `QgsVectorFileWriter` per i dettagli.

6.7.2 Direttamente dagli elementi

```
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)

(continua dalla pagina precedente)

```
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 Da un'istanza di `QgsVectorLayer`

Tra tutti i fornitori di dati supportati dalla classe `QgsVectorLayer`, concentriamoci sui layer in memoria. Il fornitore di memoria è destinato a essere usato principalmente dagli sviluppatori di plugin o di applicazioni di terze parti. Non memorizza i dati su disco, consentendo agli sviluppatori di utilizzarlo come backend veloce per alcuni layer temporanei.

Il provider supporta campi di tipo string, int e double.

Il provider di memoria supporta anche l'indicizzazione spaziale, che si attiva richiamando la funzione `createSpatialIndex()` del provider. Una volta creato l'indice spaziale, sarà possibile iterare sugli elementi all'interno di regioni più piccole in modo più rapido (poiché non è necessario analizzare tutti gli elementi, ma solo quelli nel rettangolo specificato).

Un fornitore di memoria viene creato passando `"memory"` come stringa del fornitore al costruttore `QgsVectorLayer`.

Il costruttore accetta anche un URI che definisce il tipo di geometria del layer, uno dei seguenti: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, `"MultiPolygon"` o `"None"`.

L'URI può anche specificare il sistema di riferimento delle coordinate, i campi e l'indicizzazione del provider in memoria nell'URI. La sintassi è:

crs=definizione
: Specifica il sistema di riferimento delle coordinate, dove la definizione può essere una qualsiasi delle forme accettate da `QgsCoordinateReferenceSystem.createFromString()`.

index=yes
: Specifica che il provider userà un indice spaziale

field=nome:tipo(lunghezza,precisione)
: Specifica un attributo del vettore. L'attributo ha un nome, e facoltativamente un tipo (intero, double, o string), lunghezza, e precisione. Ci sono possono essere definizioni di campo multiple

Il seguente esempio di URI incorpora tutte queste opzioni

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

Il seguente esempio di codice illustra la creazione e il popolamento di un fornitore di memoria

```
from qgis.PyQt.QtCore import QMetaType

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

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fet.setAttributes(["Johnny", 2, 0.3])
pr.addFeatures([fet])

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

Infine, verifichiamo se tutto è andato a buon fine

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

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

```
fields: 3
features: 1
extent: 10.0 10.0 10.0 10.0
F: 1 ['Johnny', 2, 0.3] <QgsPointXY: POINT(10 10)>
```

6.8 Apparenza (Simbologia) dei Vettori

Quando un vettore deve essere visualizzato, l'aspetto dei dati è dato dal **visualizzatore** e dai **simboli** associati al vettore. I simboli sono classi che si occupano del disegno di rappresentazione visiva delle geometrie, mentre i visualizzatori determinano quale simbolo sarà usato per una particolare geometria.

Il visualizzatore per un determinato layer può essere ottenuto come mostrato di seguito:

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

E con questo riferimento, esploriamolo un po".

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

```
Type: singleSymbol
```

Nella libreria principale di QGIS sono disponibili diversi tipi di visualizzatori:

Tipo	Classe	Descrizione
singleSymbol	QgsSingleSymbolRendere	Visualizza tutte le geometria con lo stesso simbolo
categorizedSym	QgsCategorizedSymbolRe	Visualizza le geometria usando un simbolo diverso per ogni categoria
graduatedSymb	QgsGraduatedSymbolRend	Visualizza le geometrie usando un simbolo diverso per ogni intervallo di valori

Potrebbero esserci anche tipi di visualizzatori personalizzati, quindi non bisogna mai dare per scontato che esistano solo questi tipi. Puoi interrogare la `QgsRendererRegistry` dell'applicazione per trovare i visualizzatori attualmente disponibili:

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

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

È possibile ottenere un dump dei contenuti di un visualizzatore in forma di testo — può essere utile per il debug.

```
renderer.dump()
```

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

6.8.1 Visualizzatore Simbolo Singolo

Puoi ottenere il simbolo usato per il visualizzatore chiamando il metodo `symbol()` e cambiarlo con il metodo `setSymbol()` (nota per gli sviluppatori C++: il visualizzatore assume la proprietà del simbolo).

Puoi modificare il simbolo utilizzato da un particolare layer vettoriale chiamando `setSymbol()` passando un'istanza del simbolo appropriato. I simboli per i layer *punto*, *linea* e *poligono* possono essere creati chiamando la funzione `createSimple()` delle classi corrispondenti `QgsMarkerSymbol`, `QgsLineSymbol` e `QgsFillSymbol`.

Il dizionario passato a `createSimple()` imposta le proprietà di stile del simbolo.

Ad esempio, puoi sostituire il simbolo utilizzato da un particolare layer **punto** chiamando `setSymbol()` passando un'istanza di `QgsMarkerSymbol`, come nel seguente esempio di codice:

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

Il ``nome"" indica la forma del marcatore e può essere uno dei seguenti:

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

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

Per ottenere l'elenco completo delle proprietà del primo livello di un'istanza di simbolo, si può seguire il codice di esempio:

```
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'}
```

Può essere utile se si desidera modificare alcune proprietà:

```
# 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 Visualizzatore Simbolo Categorizzato

Quando utilizzi un visualizzatore categorizzato, puoi interrogare e impostare l'attributo usato per la classificazione: usa i metodi `classAttribute()` e `setClassAttribute()`.

Per ottenere un elenco delle categorie

```
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>
```

Dove `value()` è il valore usato per la discriminazione tra le categorie, `label()` è un testo usato per la descrizione della categoria e `symbol()` restituisce il simbolo assegnato.

Il visualizzatore di solito memorizza anche il simbolo originale e la scala di colori utilizzati per la classificazione: metodi `sourceColorRamp()` e `sourceSymbol()`.

6.8.3 Visualizzatore Simbolo Graduato

Questo visualizzatore è molto simile al visualizzatore simbolo categorizzato descritto sopra, ma invece di un valore di attributo per classe esso lavora con intervalli di valori e quindi può essere usato solo con attributi di tipo numerico.

Per saperne di più sugli intervalli utilizzati nel visualizzatore

```
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>
```

puoi utilizzare nuovamente i metodi `classAttribute()` (per trovare il nome dell'attributo di classificazione), `sourceSymbol()` e `sourceColorRamp()`. Inoltre, esiste il metodo `mode()` che determina come sono stati creati gli intervalli: utilizzando intervalli uguali, quantili o qualche altro metodo.

Se vuoi creare un tuo renderer di simboli graduati, puoi farlo come illustrato nello frammento di codice di esempio seguente (che crea una semplice rappresentazione a due classi)

```
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)

(continua dalla pagina precedente)

```
     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 Lavorare con i simboli

Per la rappresentazione dei simboli, esiste la classe base `QgsSymbol` con tre classi derivate:

- `QgsMarkerSymbol` — per elementi punto
- `QgsLineSymbol` — per elementi linea
- `QgsFillSymbol` — per elementi poligono

Ogni simbolo consiste in uno o più livelli di simboli (classi derivate da `QgsSymbolLayer`). I livelli di simboli eseguono il rendering vero e proprio, mentre la classe simbolo serve solo come contenitore per i livelli di simboli.

Avendo un'istanza di un simbolo (ad esempio da un visualizzatore), è possibile esplorarlo: il metodo `type()` dice se si tratta di un marcatore, di una linea o di un simbolo di riempimento. Esiste un metodo `dump()` che restituisce una breve descrizione del simbolo. Per ottenere un elenco di livelli di simboli:

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

```
0: SimpleMarker
```

Per conoscere il colore del simbolo, utilizza il metodo `color()` e `setColor()` per modificarne il colore. Per i simboli marker, inoltre, puoi interrogare la dimensione e la rotazione del simbolo con i metodi `size()` e `angle()`. Per i simboli di linea, il metodo `width()` restituisce la larghezza della linea.

La dimensione e la larghezza sono in millimetri per impostazione predefinita, gli angoli sono in gradi.

Lavorare con i Livelli di Simboli

Come già detto, i livelli di simboli (sottoclassi di `QgsSymbolLayer`) determinano l'aspetto degli elementi. Esistono diverse classi di livelli di simboli di base per uso generale. È possibile implementare nuovi tipi di livelli di simboli e quindi personalizzare arbitrariamente il modo in cui gli elementi vengono rappresentati. Il metodo `layerType()` identifica in modo univoco la classe del livello di simboli: quelli di base e predefiniti sono i tipi di livelli di simboli `SimpleMarker`, `SimpleLine` e `SimpleFill`.

Puoi ottenere un elenco completo dei tipi di livelli di simboli che è possibile creare per una determinata classe di livelli di simboli con il seguente codice:

```
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
```

La classe `QgsSymbolLayerRegistry` gestisce un database di tutti i tipi di livelli di simboli disponibili.

Per accedere ai dati del livello di simboli, utilizza il metodo `properties()` che restituisce un dizionario chiave-valore di proprietà che determinano l'aspetto. Ogni tipo di livello di simboli ha un insieme specifico di proprietà che utilizza. Inoltre, ci sono i metodi generici `color()`, `size()`, `angle()` e `width()`, con le loro controparti di impostazione. Naturalmente, le dimensioni e l'angolo sono disponibili solo per i livelli dei simboli marcatore e la larghezza per i livelli dei simboli linea.

Creazione di tipi di livelli di simboli personalizzati

Immagina di voler personalizzare il modo in cui i dati vengono rappresentati. Puoi creare la tua classe di livello simbolo che disegnerà gli elementi esattamente come desideri. Ecco un esempio di un marcatore che disegna cerchi rossi con raggio specificato

```
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)
```

Il metodo `layerType()` determina il nome del livello di simboli; deve essere unico tra tutti i livelli di simboli. Il metodo `properties()` è utilizzato per la persistenza degli attributi. Il metodo `clone()` deve restituire una copia del livello di simboli con tutti gli attributi esattamente uguali. Infine ci sono i metodi di visualizzazione: `startRender()` viene chiamato prima di eseguire la visualizzazione del primo elemento, `stopRender()`

quando la visualizzazione è terminata e `renderPoint()` viene chiamato per eseguire la visualizzazione. Le coordinate dei punti sono già trasformate in coordinate di output.

Per le polilinee e i poligoni l'unica differenza è nel metodo di visualizzazione: si usa `renderPolyline()` che riceve un elenco di linee, mentre `renderPolygon()` riceve un elenco di punti sull'anello esterno come primo parametro e un elenco di anelli interni (o Nessuno) come secondo parametro.

Di solito è conveniente aggiungere un'interfaccia grafica per l'impostazione degli attributi del tipo di livello del simbolo, per consentire agli utenti di personalizzarne l'aspetto: nel caso dell'esempio precedente, possiamo consentire all'utente di impostare il raggio del cerchio. Il codice seguente implementa tale 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()"))
```

Questo widget può essere incorporato nella finestra di dialogo delle proprietà del simbolo. Quando si seleziona il tipo di livello del simbolo nella finestra di dialogo delle proprietà del simbolo, viene creata un'istanza del livello del simbolo e un'istanza del widget del livello del simbolo. Quindi chiama il metodo `setSymbolLayer()` per assegnare il livello di simboli al widget. In questo metodo il widget deve aggiornare l'interfaccia utente per riflettere gli attributi del livello di simboli. Il metodo `symbolLayer()` viene utilizzato per recuperare il livello di simboli dalla finestra di dialogo delle proprietà e utilizzarlo per il simbolo.

A ogni modifica degli attributi, il widget deve emettere il messaggio `changed()` per consentire alla finestra di dialogo delle proprietà di aggiornare l'anteprima del simbolo.

Ora manca solo il collante finale: rendere QGIS consapevole di queste nuove classi. Questo viene fatto aggiungendo il livello dei simboli al registro. È possibile utilizzare il livello di simboli anche senza aggiungerlo al registro, ma alcune funzionalità non funzioneranno: ad esempio, il caricamento dei file di progetto con i livelli di simboli personalizzati o l'impossibilità di modificare gli attributi del layer nella GUI.

Dovremo creare i metadati per il livello dei simboli

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

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):
```

(continues on next page)

(continua dalla pagina precedente)

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

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

fslmetadata = FooSymbolLayerMetadata()
```

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

Devi passare il tipo di livello (lo stesso restituito dal livello) e il tipo di simbolo (marcatore/linea/riempimento) al costruttore della classe padre. Il metodo `createSymbolLayer()` si occupa di creare un'istanza di livello di simboli con gli attributi specificati nel dizionario *props*. E c'è il metodo `createSymbolLayerWidget()` che restituisce il widget delle impostazioni per questo tipo di livello di simboli.

L'ultimo passo consiste nell'aggiungere questo livello di simboli al registro — e abbiamo finito.

6.8.5 Creazione di visualizzatori personalizzati

Potrebbe essere utile creare una nuova implementazione del visualizzatore se vuoi personalizzare le regole di selezione dei simboli per la visualizzazione degli elementi. Alcuni casi d'uso in cui si vorrebbe farlo: il simbolo è determinato da una combinazione di campi, la dimensione dei simboli cambia a seconda della scala corrente, ecc.

Il codice seguente mostra un semplice visualizzatore personalizzato che crea due simboli di marcatori e ne sceglie a caso uno per ogni elemento

```
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)
```

Il costruttore della classe genitore `QgsFeatureRenderer` ha bisogno di un nome di visualizzatore (che deve essere unico tra i visualizzatori). Il metodo `symbolForFeature()` è quello che decide quale simbolo sarà usato per un particolare elemento. `startRender()` e `stopRender()` si occupano dell'inizializzazione/finalizzazione della rappresentazione dei simboli. Il metodo `usedAttributes()` può restituire un elenco di nomi di campi che il visualizzatore si aspetta siano presenti. Infine, il metodo `clone()` dovrebbe restituire una copia del visualizzatore.

Come per i livelli di simboli, è possibile collegare un'interfaccia grafica per la configurazione del visualizzatore. Deve essere derivata da `QgsRendererWidget`. Il codice di esempio seguente crea un pulsante che consente all'utente di impostare il primo simbolo

```
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
```

Il costruttore riceve istanze del livello attivo (`QgsVectorLayer`), dello stile globale (`QgsStyle`) e del visualizzatore corrente. Se non c'è un visualizzatore o se il visualizzatore ha un tipo diverso, verrà sostituito con il nostro nuovo visualizzatore, altrimenti verrà utilizzato il visualizzatore corrente (che ha già il tipo di cui abbiamo bisogno). Il contenuto del widget deve essere aggiornato per mostrare lo stato attuale del visualizzatore. Quando la finestra di dialogo del visualizzatore viene accettata, viene richiamato il metodo `renderer()` del widget per ottenere il visualizzatore corrente, che verrà assegnato al livello.

L'ultima parte mancante sono i metadati del visualizzatore e la sua registrazione nel registro, altrimenti il caricamento dei livelli con il visualizzatore non funzionerà e l'utente non sarà in grado di selezionarlo dall'elenco dei visualizzatori. Terminiamo l'esempio di RandomRenderer

```
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):
```

(continues on next page)

(continua dalla pagina precedente)

```
    return RandomRendererWidget(layer, style, renderer)

rrmetadata = RandomRendererMetadata()
```

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

Come per i livelli di simboli, il costruttore dell'abstract dei metadati si aspetta il nome del visualizzatore, il nome visibile per gli utenti e, facoltativamente, il nome dell'icona del visualizzatore. Il metodo `createRenderer()` passa un'istanza `QDomElement` che può essere usata per ripristinare lo stato del visualizzatore dall'albero DOM. Il metodo `createRendererWidget()` crea il widget di configurazione. Non è necessario che sia presente o può restituire `None` se il visualizzatore non è dotato di interfaccia grafica.

Per associare un'icona al visualizzatore, puoi assegnarla nel costruttore `QgsRendererAbstractMetadata` come terzo parametro (opzionale) — il costruttore della classe base nella funzione RandomRendererMetadata `__init__()` diventa

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

L'icona può anche essere associata in qualsiasi momento successivo usando il metodo `setIcon()` della classe dei metadati. L'icona può essere caricata da un file (come mostrato sopra) o da una risorsa Qt (PyQt5 include il compilatore .qrc per Python).

6.9 Ulteriori argomenti

****DA FARE: ****

- creazione/modifica di simboli
- lavorare con gli stili (`QgsStyle`)
- lavorare con la scala di colori (`QgsColorRamp`)
- esplorare i registri dei livelli di simboli e dei visualizzatori

CAPITOLO 7

Gestione della Geometria

Suggerimento: I frammenti di codice di questa pagina necessitano delle seguenti importazioni se sei è al di fuori della console pyqgis:

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

Punti, linee e poligoni che rappresentano un elemento spaziale sono comunemente indicati come geometrie. In QGIS sono rappresentati con la `QgsGeometry` class.

Alcune volte una geometria é effettivamente una collezione di geometrie (parti singole) piú semplici. Se contiene un tipo di geometria semplice, la chiameremo punti multipli, string multi linea o poligoni multipli. Ad esempio, un Paese formato da piú isole puó essere rappresentato come un poligono multiplo.

Le coordinate delle geometrie possono essere in qualsiasi sistema di riferimento delle coordinate (CRS). Quando si estraggono delle caratteristiche da un vettore, le geometrie associate avranno le coordinate nel CRS del vettore.

La descrizione e le specifiche di tutte le possibili geometrie e relazioni sono disponibili per dettagli avanzati nel documento OGC Simple Feature Access Standards .

7.1 Costruzione della Geometria

PyQGIS offre diverse opzioni per la creazione di una geometria:

- dalle coordinate

```
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)
```

  Le coordinate vengono fornite utilizzando la classe `QgsPoint` o la classe `QgsPointXY`. La differenza tra queste classi è che `QgsPoint` supporta le dimensioni M e Z.

  Una Polilinea (Linestring) è rappresentata da un elenco di punti.

  Un Poligono è rappresentato da un elenco di anelli lineari (cioè di linee chiuse). Il primo anello è l'anello esterno (confine), mentre eventuali anelli successivi sono buchi nel poligono. Da notare che, a differenza di alcuni programmi, QGIS chiude l'anello per voi, quindi non è necessario duplicare il primo punto come ultimo.

  Le geometrie a parti multiple vanno ad un livello successivo: punti multipli é una lista di punti, una stringa multi linea é una linea di linee ed un poligono multiplo é una lista di poligoni.

- da well-known text (WKT)

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

- da well-known binary (WKB)

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

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

7.2 Accedere alla Geometria

Per prima cosa, devi trovare il tipo di geometria. Il metodo `wkbType()` è quello da utilizzare. Restituisce un valore dell'enumerazione `QgsWkbTypes.Type`.

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

In alternativa, si può usare il metodo `type()` che restituisce un valore dell'enumerazione `QgsWkbTypes.GeometryType`.

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

Puoi usare la funzione `displayString()` per ottenere un tipo di geometria leggibile in chiaro.

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

Esiste anche una funzione di aiuto `isMultipart()` per scoprire se una geometria è multipart o meno.

Per estrarre informazioni da una geometria esistono funzioni di selezione per ogni tipologia di vettore. Ecco un esempio di utilizzo di queste funzioni di selezione:

```
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)>]]
```

Nota: Le tuple (x,y) non sono tuple reali, ma oggetti `QgsPoint`, i cui valori sono accessibili con i metodi `x()` e `y()`.

Per le geometrie multiparte esistono funzioni di accesso simili: `asMultiPoint()`, `asMultiPolyline()` e `asMultiPolygon()`.

È possibile iterare su tutte le parti di una geometria, indipendentemente dal tipo di geometria. Ad esempio

```
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)
```

È anche possibile modificare ciascuna parte della geometria usando il metodo `QgsGeometry.parts()`.

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

(continues on next page)

(continua dalla pagina precedente)

```
8
9  print(geom.asWkt())
```

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

7.3 Predicati ed Operazioni delle Geometrie

QGIS utilizza la libreria GEOS per operazioni geometriche avanzate come i predicati geometrici (`contains()`, `intersects()`, ...) e operazioni di insieme (`combine()`, `difference()`, ...). Può anche calcolare le proprietà geometriche delle geometrie, come l'area (nel caso dei poligoni) o le lunghezze (per poligoni e linee).

Vediamo un esempio che combina l'iterazione degli elementi in un dato layer e l'esecuzione di alcuni calcoli geometrici basati sulle loro geometrie. Il codice seguente calcolerà e stamperà l'area e il perimetro di ogni countrie nel layer ``countries"" del nostro progetto QGIS.

Il codice seguente presuppone che `layer` sia un oggetto `QgsVectorLayer` che ha una tipologia Poligono.

```
1  # let's access the 'countries' layer
2  layer = QgsProject.instance().mapLayersByName('countries')[0]
3
4  # let's filter for countries that begin with Z, then get their features
5  query = '"name" LIKE \'Z%\''
6  features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))
7
8  # now loop through the features, perform geometry computation and print the results
9  for f in features:
10   geom = f.geometry()
11   name = f.attribute('NAME')
12   print(name)
13   print('Area: ', geom.area())
14   print('Perimeter: ', geom.length())
```

```
1  Zambia
2  Area:  62.82279065343119
3  Perimeter:  50.65232014052552
4  Zimbabwe
5  Area:  33.41113559136517
6  Perimeter:  26.608288555013935
```

Ora hai calcolato e stampato le aree e i perimetri delle geometrie. Tuttavia, puoi subito notare che i valori sono strani. Questo perché le aree e i perimetri non tengono conto dei SR quando vengono calcolati con i metodi `area()` e `length()` della classe `QgsGeometry`. Per un calcolo più preciso di aree e distanze, occorre utilizzare la classe `QgsDistanceArea`, che può eseguire calcoli basati su ellissoidi:

Il codice seguente presuppone che `layer` sia un oggetto `QgsVectorLayer` che ha una tipologia Poligono.

```
1  d = QgsDistanceArea()
2  d.setEllipsoid('WGS84')
3
4  layer = QgsProject.instance().mapLayersByName('countries')[0]
5
6  # let's filter for countries that begin with Z, then get their features
7  query = '"name" LIKE \'Z%\''
8  features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))
9
10 for f in features:
```

(continues on next page)

(continua dalla pagina precedente)

```
  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
```

In alternativa, potresti voler conoscere la distanza tra due punti.

```
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))
```

Puoi trovare molti esempi di algoritmi che sono inclusi in QGIS ed utilizzare questi metodi per analizzare e trasformare i dati vettoriali. Di seguito i link al codice di alcuni di questi.

- Distanza e area utilizzando la classe `QgsDistanceArea`: Distance matrix algorithm.
- Lines to polygons algorithm

CAPITOLO 8

Supporto alle proiezioni

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

8.1 Sistemi di riferimento delle coordinate

I sistemi di riferimento delle coordinate (SR) sono incapsulati dalla classe `QgsCoordinateReferenceSystem`. Le istanze di questa classe possono essere create in diversi modi:

- Specificare SR con il suo ID

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

```
True
```

QGIS supporta diversi identificatori SR con i seguenti formati:

  - `EPSG:<code>` — ID assegnato dall'organizzazione EPSG - gestito con `createFromOgcWms()`
  - `POSTGIS:<srid>`— ID utilizzato nei database PostGIS - gestito con `createFromSrid()`
  - `INTERNAL:<srsid>` — ID utilizzato nel database interno di QGIS - gestito con `createFromSrsId()`
  - `PROJ:<proj>` - gestito con `createFromProj()`
  - `WKT:<wkt>` - gestito con `createFromWkt()`

Se non viene specificato alcun prefisso, viene assunta la definizione WKT.

- specifica il SR con il suo well-known text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.
↪257223563]],' \
      'PRIMEM["Greenwich", 0.0], UNIT["degree",0.0174532925199433295],' \
      'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
print(crs.isValid())
```

```
True
```

- crea un SR non valido e poi utilizzare una delle funzioni `create*` per inizializzarlo. Nell'esempio seguente si utilizza una stringa Proj per inizializzare la proiezione.

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

```
True
```

È opportuno verificare se la creazione (cioè la ricerca nel database) del SR è avvenuta con successo: `isValid()` deve restituire `True`.

Si noti che per l'inizializzazione dei sistemi di riferimento spaziali QGIS deve cercare i valori appropriati nel suo database interno `srs.db`. Pertanto, se creai un'applicazione indipendente, devi impostare correttamente i percorsi con `QgsApplication.setPrefixPath()`, altrimenti non riuscirà a trovare il database. Se esegui i comandi dalla console QGIS Python o sviluppi un plugin, non è importante: tutto è già stato impostato.

Accesso alle informazioni del sistema di riferimento spaziale:

```
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())
```

In uscita:

```
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 Trasformazione SR

Puoi effettuare trasformazioni tra sistemi di riferimento spaziali diversi utilizzando la classe `QgsCoordinateTransform`. Il modo più semplice per usarla è creare un SR di origine e uno di destinazione e costruire un'istanza `QgsCoordinateTransform` con essi e il progetto corrente. Quindi basta chiamare ripetutamente la funzione `transform()` per eseguire la trasformazione. Per impostazione predefinita esegue la trasformazione in avanti, ma è in grado di eseguire anche la trasformazione inversa.

```
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)
```

In uscita:

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

CAPITOLO 9

Utilizzare l'Area di Mappa

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

```
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,
)
```

Il widget Mappa è probabilmente il widget più importante di QGIS, perché mostra la mappa composta da layer sovrapposti e consente l'interazione con la mappa e i layer. L'area di disegno della mappa mostra sempre una parte della mappa definita dall'estensione corrente dell'area. L'interazione avviene attraverso l'uso di **strumenti mappa**: ci sono strumenti per la panoramica, lo zoom, l'identificazione dei layer, la misurazione, l'editing vettoriale e altri. Come in altri programmi di grafica, c'è sempre uno strumento attivo e l'utente può passare da uno strumento all'altro.

L'area di mappa è implementata con la classe `QgsMapCanvas` nel modulo qgis.gui. L'implementazione è basata sul framework Qt Graphics View. Questo framework fornisce generalmente una superficie e una vista in cui vengono

inseriti elementi grafici personalizzati con cui l'utente può interagire. Si presuppone che si abbia abbastanza familiarità con Qt da comprendere i concetti di scena grafica, vista e oggetti. In caso contrario, si consiglia di leggere overview of the framework.

Ogni volta che la mappa viene spostata, ingrandita o rimpicciolita (o qualsiasi altra azione che attivi un refresh), la mappa viene nuovamente visualizzata all'interno dell'estensione corrente. I layer vengono resi in un'immagine (utilizzando la classe `QgsMapRendererJob`) e l'immagine viene visualizzata sulla area mappa. La classe `QgsMapCanvas` controlla anche il refresh della mappa visualizzata. Oltre a questo elemento che funge da sfondo, ci possono essere altri **oggetti della area mappa**.

I tipici elementi della area mappa sono gli elastici (usati per la misurazione, la modifica vettoriale e così via) o i marcatori di vertici. Gli oggetti mappa sono solitamente utilizzati per fornire un feedback visivo agli strumenti di mappa; ad esempio, quando si crea un nuovo poligono, lo strumento di mappa crea un elemento mappa ad elastico che mostra la forma corrente del poligono. Tutti gli oggetti in mappa delle mappe sono sottoclassi di `QgsMapCanvasItem` che aggiungono alcune funzionalità agli oggetti `QGraphicsItem` di base.

In sintesi, l'architettura dell'area mappa si compone di tre concetti:

- area mappa — per la visualizzazione della mappa
- oggetti dell'area mappa — oggetti aggiuntivi che possono essere visualizzati sull'area mappa
- strumenti mappa — per interagire con la area mappa

9.1 Incorporare l'area mappa

L'area di disegno della mappa è un widget come qualsiasi altro widget di Qt, quindi utilizzarlo è semplice come crearlo e visualizzarlo.

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

Produce una finestra indipendente con l'area di disegno della mappa. Può anche essere incorporata in un widget o in una finestra esistente. Quando si usano i file `.ui` e Qt Designer, posizionare un `QWidget` sul modulo e promuoverlo a una nuova classe: impostare `QgsMapCanvas` come nome della classe e impostare `qgis.gui` come file di intestazione. L'utilità `pyuic5` se ne occuperà. Questo è un modo molto comodo di incorporare l'area mappa. L'altra possibilità è scrivere manualmente il codice per costruire l'area di disegno della mappa e altri widget (come figli di una finestra principale o di una finestra di dialogo) e creare un layout.

Per impostazione predefinita, l'area mappa ha uno sfondo nero e non utilizza l'anti-aliasing. Per impostare uno sfondo bianco e abilitare l'anti-aliasing per una visualizzazione ottimale

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

(Nel caso ve lo stiate chiedendo, `Qt` proviene dal modulo `PyQt.QtCore` e `Qt.white` è una delle istanze predefinite di `QColor`).

Ora è il momento di aggiungere alcuni layer della mappa. Per prima cosa apriremo un layer e lo aggiungeremo al progetto corrente. Quindi si imposterà l'estensione della area mappa e si imposterà l'elenco dei layer per tale area.

```
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())
```

(continues on next page)

(continua dalla pagina precedente)

```
10
11 # set the map canvas layer set
12 canvas.setLayers([vlayer])
```

Dopo aver eseguito questi comandi, l'area di disegno dovrebbe mostrare il layer caricato.

9.2 Bande elastiche e marcatori di vertice

Per mostrare alcuni dati aggiuntivi sopra la mappa nell'area di disegno della mappa, usa oggetti di mappa. Puoi creare classi di oggetti mappa personalizzate (descritte di seguito), tuttavia esistono due classi di oggetti mappa utili per comodità: `QgsRubberBand` per disegnare polilinee o poligoni e `QgsVertexMarker` per disegnare punti. Entrambi funzionano con le coordinate della mappa, quindi la forma viene spostata/ridimensionata automaticamente quando l'area mappa viene spostata o ingrandita.

Per mostrare una polilinea:

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

Per mostrare un poligono

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

Si noti che i punti per il poligono non sono un semplice elenco: si tratta infatti di un elenco di anelli contenenti anelli lineari del poligono: il primo anello è il bordo esterno, gli altri anelli (facoltativi) corrispondono ai fori del poligono.

Le bande ad elastico consentono una certa personalizzazione, ovvero la modifica del colore e della larghezza della linea.

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

Gli oggetti dell'area di disegno mappa sono legati alla scena dell'area di disegno. Per nasconderli temporaneamente (e mostrarli di nuovo), utilizza le combinazioni `hide()` e `show()`. Per rimuovere completamente l'elemento, devi rimuoverlo dalla scena del dell'area di disegno

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

(in C++ è possibile cancellare semplicemente l'oggetto, ma in Python `del r` cancellerebbe solo il riferimento e l'oggetto continuerebbe a esistere, essendo di proprietà dell'area mappa).

Gli elastici possono essere usati anche per disegnare punti, ma la classe `QgsVertexMarker` è più adatta a questo scopo (`QgsRubberBand` disegnerebbe solo un rettangolo attorno al punto desiderato).

Puoi utilizzare il marcatore di vertici in questo modo:

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

Questo disegnerà una croce rossa sulla posizione **[10,45]**. È possibile personalizzare il tipo di icona, le dimensioni, il colore e la larghezza della penna.

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

Per nascondere temporaneamente i marcatori di vertici e rimuoverli dall" area di disegno, utilizza gli stessi metodi utilizzati per gli elastici.

9.3 Utilizzo degli strumenti di mappa con con l'area mappa

L'esempio seguente costruisce una finestra che contiene una area mappa e gli strumenti di base per il panning e lo zoom della mappa. Vengono create azioni per l'attivazione di ogni strumento: la panoramica viene eseguita con `QgsMapToolPan`, lo zoom in/out con una coppia di istanze `QgsMapToolZoom`. Le azioni sono impostate come controllabili e successivamente assegnate agli strumenti, per consentire la gestione automatica dello stato controllato/non controllato delle azioni: quando uno strumento di mappa viene attivato, la sua azione viene contrassegnata come selezionata e l'azione dello strumento di mappa precedente viene deselezionata. Gli strumenti delle mappe vengono attivati utilizzando il metodo `setMapTool()`.

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

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

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

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

        self.setCentralWidget(self.canvas)

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

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

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

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

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

        self.pan()

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

    def zoomOut(self):
```

(continues on next page)

(continua dalla pagina precedente)

```
        self.canvas.setMapTool(self.toolZoomOut)

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

Puoi provare il codice precedente nell'editor della console di Python. Per richiamare la finestra dell'are di disegno mappa, aggiungi le righe seguenti per istanziare la classe `MyWnd`. Esse visualizzeranno il layer attualmente selezionato sulla area di disegno appena creata

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

9.3.1 Seleziona un elemento usando QgsMapToolIdentifyFeature

Puoi utilizzare lo strumento mappa `QgsMapToolIdentifyFeature` per chiedere all'utente di selezionare un elemento che sarà inviato a una funzione di callback.

```
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 Aggiungere elementi al menu contestuale dell'area di disegno della mappa

L'interazione con il disegno della mappa può essere fatta anche attraverso gli elementi che si possono aggiungere al suo menu contestuale, utilizzando il segnale `contextMenuAboutToShow`.

Il codice seguente aggiunge l'azione *Mio menu* ► *Mia Azione* accanto alle voci predefinite quando fai clic con il pulsante destro del mouse sulla area di disegno della mappa.

```
# 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 Scrivere Strumenti Mappa Personalizzati

Puoi scrivere strumenti personalizzati, per implementare un comportamento personalizzato alle azioni eseguite dagli utenti sull'area di disegno della mappa.

Gli strumenti di mappa devono essere ereditati dalla classe `QgsMapTool`, o da qualsiasi classe derivata, e devono essere selezionati come strumenti attivi nell'area di disegno utilizzando il metodo `setMapTool()`, come abbiamo già visto.

Ecco un esempio di strumento mappa che consente di definire un'estensione rettangolare facendo clic e trascinando sull'area di disegno. Quando il rettangolo è definito, ne stampa le coordinate di confine nella console. Utilizza gli elementi a elastico descritti in precedenza per mostrare il rettangolo selezionato mentre viene definito.

```
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
```

(continues on next page)

(continua dalla pagina precedente)

```
    self.rubberBand.show()

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

      return QgsRectangle(self.startPoint, self.endPoint)

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

9.5 Scrittura Oggetti Personalizzati Disegno Mappa

Ecco un esempio di un oggetto personalizzato che disegna un cerchio sull'area di disegno mappa:

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

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

  def center(self):
    return self.center

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

  def size(self):
    return self.size

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

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


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

CAPITOLO 10

Visualizzazione e Stampa di una Mappa

Suggerimento: I frammenti di codice in questa pagina hanno bisogno dei seguenti 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,
)
```

Ci sono generalmente due approcci quando i dati di input devono essere visualizzati come mappe: o farlo in modo rapido usando *QgsMapRendererJob* o produrre un output più preciso componendo la mappa con la classe `QgsLayout`.

10.1 Visualizzazione Semplice

La visualizzazione avviene creando un oggetto `QgsMapSettings` per definire le impostazioni di visualizzazione e costruendo poi un `QgsMapRendererJob` con tali impostazioni. Quest'ultimo viene poi utilizzato per creare l'immagine risultante.

Ecco un esempio:

```
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 Visualizzare layer con diversi SR

Se hai più di un layer e questi hanno un SR diverso, il semplice esempio precedente probabilmente non funzionerà: per ottenere i valori giusti dai calcoli di estensione devi impostare esplicitamente il SR di destinazione

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

10.3 Risultato con layout di stampa

Il layout di stampa è uno strumento molto utile se vuoi ottenere un output più sofisticato rispetto alla semplice rappresentazione mostrata sopra. Puoi creare layout cartografici complessi composti da viste della mappa, etichette, legende, tabelle e altri elementi solitamente presenti nelle mappe cartacee. I layout possono essere esportati in PDF, SVG, immagini raster o stampati direttamente su una stampante.

Il layout è costituito da un gruppo di classi. Tutte appartengono alla libreria principale. L'applicazione QGIS ha una comoda interfaccia grafica per il posizionamento degli elementi, anche se non è disponibile nella libreria GUI. Se non hai familiarità con il framework Qt Graphics View, ti consigliamo di consultare ora la documentazione, perché il layout si basa su di esso.

La classe principale del layout è la classe `QgsLayout`, che deriva dalla classe Qt QGraphicsScene. Creiamo un'istanza di questa classe:

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

Questo metodo inizializza il layout con alcune impostazioni predefinite, in particolare aggiungendo una pagina A4 vuota al layout. Puoi creare layout senza chiamare il metodo `initializeDefaults()`, ma dovrai occuparti personalmente di aggiungere pagine al layout.

Il codice precedente crea un layout «temporaneo» che non è visibile nella GUI. Può essere utile, ad esempio, per aggiungere rapidamente alcuni elementi ed esportare senza modificare il progetto stesso né esporre le modifiche all'utente. Se vuoi che il layout venga salvato/ripristinato insieme al progetto e sia disponibile nel gestore dei layout, aggiungi:

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

Ora possiamo aggiungere vari elementi (mappa, etichetta, …) al layout. Tutti questi oggetti sono rappresentati da classi che ereditano dalla classe base `QgsLayoutItem`.

Ecco una descrizione di alcuni dei principali oggetti di layout che possono essere aggiunti a un layout.

- map — Qui creiamo una mappa di dimensioni personalizzate e visualizziamo l'area mappa della mappa corrente.

```
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 — permetta la visualizzazione di etichette. É possibile modificarne il carattere, colore, allineamento e margine

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

- legenda

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

- barra di scala

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

- forma basata su nodi

```
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)
```

Una volta aggiunto un oggetto al layout, è possibile spostarlo e ridimensionarlo:

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

Per impostazione predefinita, attorno a ciascun elemento viene disegnata una cornice. Puoi rimuoverla come segue:

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

Oltre a creare gli oggetti del layout a mano, QGIS supporta i modelli di layout, che sono essenzialmente composizioni con tutti i loro oggetti salvati in un file .qpt (con sintassi XML).

Una volta che la composizione è pronta (gli oggetti del layout sono stati creati e aggiunti alla composizione), possiamo procedere alla produzione di un output raster e/o vettoriale.

10.3.1 Controllo della validità del layout

Un layout è costituito da un insieme di elementi interconnessi e può accadere che queste connessioni vengano interrotte durante le modifiche (una legenda collegata a una mappa rimossa, un elemento immagine con un file sorgente mancante, …) o che si vogliano applicare vincoli personalizzati agli elementi del layout. La `QgsAbstractValidityCheck` aiuta a raggiungere questo obiettivo.

Un controllo di base si presenta come segue:

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

Ecco un controllo che genera un avviso ogni volta che un elemento della mappa di layout è impostato sulla proiezione web mercator:

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

  return results
```

Ed ecco un esempio più complesso, che genera un avviso se uno qualsiasi degli elementi della mappa di layout è impostato su un SR valido solo al di fuori dell'estensione mostrata in quell'elemento della mappa:

```
@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 Esporta il layout

Per esportare un layout, è necessario utilizzare la classe `QgsLayoutExporter`.

```
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())
```

Usa `exportToSvg()` o `exportToImage()` nel caso in cui tui voglia esportare rispettivamente in un file SVG o in un file immagine invece che in un file PDF.

10.3.3 Esportare un layout atlante

Se vuoi esportare tutte le pagine da un layout che ha l'opzione atlante configurata e abilitata, devi usare il metodo `atlas()` nell'esportatore (`QgsLayoutExporter`) con piccole modifiche. Nell'esempio seguente, le pagine sono esportate in immagini PNG:

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

Da notare che gli output saranno salvati nella cartella del percorso di base, utilizzando l'espressione del nome del file di output configurata su atlante.

CAPITOLO 11

Espressioni, Filtraggio e Calcolo di Valori

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

QGIS ha un qualche supporto per l'analisi di espressioni di tipo SQL. È supportato solo un piccolo sottoinsieme della sintassi SQL. Le espressioni possono essere valutate come predicati booleani (restituendo `True` o `False`) o come funzioni (restituendo un valore scalare). Per un elenco completo delle funzioni disponibili, vedere vector_expressions nel Manuale utente.

Sono supportati tre tipi base:

- numero – sia numeri interi che decimali, e.g. `123`, `3.14`
- stringa – devono essere racchiuse tra apici singoli: `'hello world'`
- riferimento a colonna – durante la valutazione, il riferimento é sostituito con il valore del campo. I nomi non sono racchiusi tra apici.

Sono disponibili le seguenti operazioni:

- operatori aritmetici: `+`, `-`, `*`, `/`, `^`
- parentesi: per forzare la precedenza tra operatori: `(1 + 1) * 3`
- somma e sottrazione unari: `-12`, `+5`

- funzioni matematiche: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- funzioni di conversione: `to_int`, `to_real`, `to_string`, `to_date`
- funzioni sulla geometria: `$area`, `$length`
- funzioni di manipolazione della geometria: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

Sono supportati i seguenti predicati:

- comparazione: =, !=, >, >=, <, <=
- pattern matching: `LIKE` (usando % e _), ~ (espressioni regolari)
- predicati logici: `AND`, `OR`, `NOT`
- controllo di valori NULL: `IS NULL`, `IS NOT NULL`

Esempi di predicati:

- `1 + 2 = 3`
- `sin(angolo) > 0`
- `'Hello' LIKE 'He%'`
- `(x > 10 AND y > 10) OR z = 0`

Esempi di espressioni scalari:

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

11.1 Analisi di Espressioni

L'esempio seguente mostra come verificare se una determinata espressione può essere analizzata correttamente:

```
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 Valutazione di Espressioni

Le espressioni possono essere utilizzate in diversi contesti, ad esempio per filtrare gli elementi o per calcolare i valori di nuovi campi. In ogni caso, l'espressione deve essere valorizzata. Ciò significa che il suo valore viene calcolato eseguendo le fasi di calcolo specificate, che possono andare dalla semplice aritmetica alle espressioni aggregate.

11.2.1 Espressioni Base

Questa espressione di base valuta una semplice operazione aritmetica:

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

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

L'espressione può essere utilizzata anche per il confronto, fornendo una risposta pari a 1 (`True`) o a 0 (`False`).

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

11.2.2 Espressioni con geometrie

Per valutare un'espressione rispetto a un elemento, è necessario creare un oggetto `QgsExpressionContext` e passarlo alla funzione evaluate, per consentire all'espressione di accedere ai valori dei campi dell'elemento.

L'esempio seguente mostra come creare un elemento con un campo chiamato «Column» e come aggiungere questo elemento al contesto dell'espressione.

```
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
```

Di seguito è riportato un esempio più completo di come utilizzare le espressioni nel contesto di un layer vettoriale, al fine di calcolare nuovi valori di campo:

```
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)

(continua dalla pagina precedente)

```
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 Filtrare un layer con le espressioni

L'esempio seguente puó essere usato per filtrare un layer e restituire qualsiasi geometria che soddisfi il predicato.

```
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)

(continua dalla pagina precedente)

```
    matches += 1

print(matches)
```

```
7
```

11.3 Gestione degli errori delle espressioni

Gli errori relativi alle espressioni possono verificarsi durante l'analisi o la valutazione delle stesse:

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

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

CAPITOLO 12

Leggere e Memorizzare Impostazioni

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

Molte volte è utile per un plugin salvare alcune variabili in modo tale che l'utente non debba inserirle o selezionarle di nuovo la volta successiva che il plugin è eseguito.

Queste variabili possono essere salvate e recuperate con l'aiuto delle API di Qt e QGIS. Per ogni variabile, dovresti scegliere una chiave che sarà usata per accedere alla variabile — per il colore preferito dell'utente potresti usare la chiave «favourite_color» o una qualunque altra stringa significativa. È raccomandabile dare una qualche struttura alla denominazione delle chiavi.

Possiamo distinguere tra diversi tipi di impostazioni:

- **impostazioni globali** — sono legate all'utente di una particolare macchina. QGIS stesso memorizza molte impostazioni globali, ad esempio la dimensione della finestra principale o la tolleranza di aggancio predefinita. Le impostazioni sono gestite dalla classe `QgsSettings`, ad esempio attraverso i metodi `setValue()` e `value()`.

  Qui puoi vedere un esempio di utilizzo di questi metodi.

```
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)
```

(continues on next page)

(continua dalla pagina precedente)

```
  nonexistent = s.value("myplugin/nonexistent", None)
  print(mytext)
  print(myint)
  print(myreal)
  print(nonexistent)
```

Il secondo parametro del metodo `value()` è opzionale e specifica il valore predefinito che viene restituito se non esiste un valore precedente per il nome dell'impostazione fornita.

Per un metodo di preconfigurazione dei valori predefiniti delle impostazioni globali attraverso il file `qgis_global_settings.ini`, vedere deploying_organization per ulteriori dettagli.

- **Impostazioni di progetto** — variano tra i diversi progetti e quindi sono collegate a un file di progetto. Il colore dello sfondo della area mappa o il sistema di riferimento delle coordinate (SR) di destinazione ne sono un esempio — lo sfondo bianco e il WGS84 potrebbero essere adatti a un progetto, mentre lo sfondo giallo e la proiezione UTM sono migliori per un altro.

  Segue un esempio di utilizzo.

```
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)
```

Come puoi vedere, il metodo `writeEntry()` viene utilizzato per molti tipi di dati (interi, stringhe, elenchi), ma esistono diversi metodi per leggere il valore dell'impostazione e per ogni tipo di dati deve essere selezionato quello corrispondente.

- **impostazioni layer mappa** — queste impostazioni sono relative a una particolare istanza di un layer mappa con un progetto. Sono *non* collegate alla sorgente dati sottostante di un layer, quindi se crei due istanze di layer di mappa di uno shapefile, non condivideranno le impostazioni. Le impostazioni sono memorizzate all'interno del file di progetto, quindi se l'utente apre nuovamente il progetto, le impostazioni relative al layer saranno di nuovo presenti. Il valore di una determinata impostazione viene recuperato con il metodo `customProperty()` e può essere impostato con il metodo `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")
```

CAPITOLO 13

Comunicare con l'utente

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

from qgis.gui import (
    QgsMessageBar,
)

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

Questa sezione mostra alcuni metodi ed elementi che dovrebbero essere usati per comunicare con l'utente, in modo da mantenere la consistenza nell'interfaccia utente.

13.1 Mostrare i messaggi. La classe QgsMessageBar

Utilizzare il box dei messaggi potrebbe essere una cattiva idea dal punto di vista dell'esperienza utente. Solitamente, per mostrare un messaggio di informazione o di errore/avvertimento, la barra dei messaggi di QGIS é l'opzione migliore.

Utilizzando il riferimento all'oggetto interfaccia di QGIS, puoi mostrare un messaggio nella barra dei messaggi utilizzando il seguente codice

```
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
```

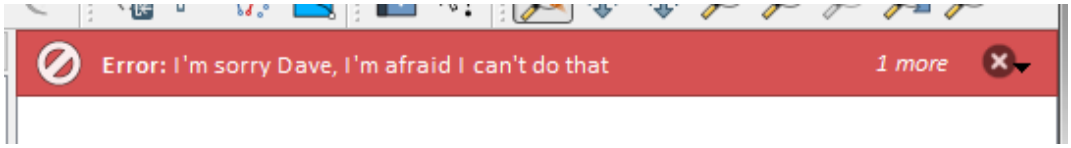


Fig. 13.1: Barra dei messaggi di QGIS

Puoi impostare una durata per visualizzarlo per un tempo limitato

```
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
```

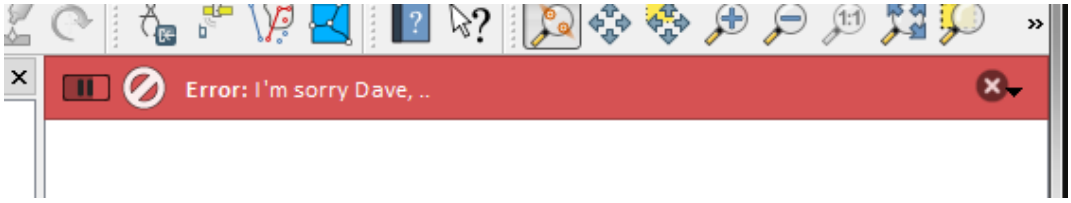


Fig. 13.2: Barra dei messaggi di QGIS con timer

Gli esempi precedenti mostrano una barra di errore, ma il parametro `level` può essere usato per creare messaggi di avviso o di informazione, usando l'enumerazione `Qgis.MessageLevel`. Puoi utilizzare fino a 4 livelli diversi:

0. Info
1. Warning
2. Critical
3. Success

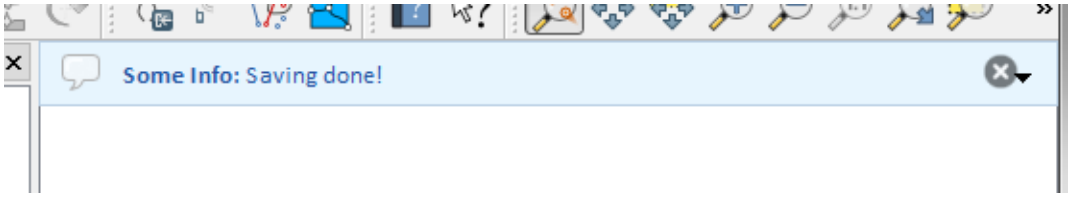


Fig. 13.3: Barra dei messaggi di QGIS (informazioni)

I widget possono essere aggiunti alla barra dei messaggi, ad esempio il pulsante per mostrare piú informazioni

```
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
```

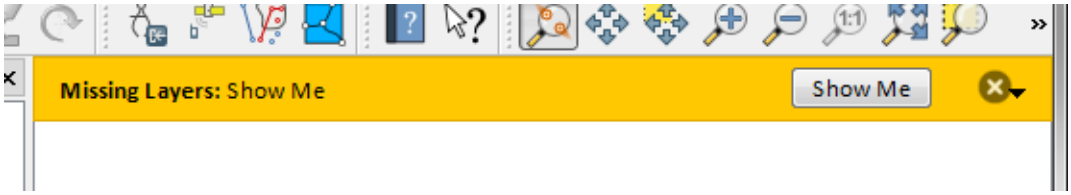


Fig. 13.4: Barra dei messaggi di QGIS con un pulsante

É possibile usare una barra dei messaggi nella propria finestra di dialogo senza dover mostrare una finestra di messaggi, o nel caso in cui non abbia senso mostrarla nella finestra principale di QGIS.

```
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: Barra dei messaggi di QGIS in una finestra di dialogo personalizzata

13.2 Mostrare l'avanzamento

Le barre di avanzamento si possono mettere anche nella barra dei messaggi di QGIS, dato che, come abbiamo visto, accetta i widget. Di seguito un esempio che potrete provare nella console.

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

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

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

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

Inoltre, puoi utilizzare la barra di stato integrata per segnalare i progressi, come nell'esempio successivo:

```
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 Logging

In QGIS sono disponibili tre diversi tipi di registrazione per registrare e salvare tutte le informazioni sull'esecuzione del tuo codice. Ognuno di essi ha una posizione di output specifica. Si consiglia di utilizzare il metodo di registrazione corretto per il tuo scopo:

- `QgsMessageLog` è per i messaggi che comunicano i problemi all'utente. L'output di QgsMessageLog è mostrato nel pannello dei messaggi di log.
- Il modulo **logging** integrato in Python serve per il debug a livello dell'API Python di QGIS (PyQGIS). È consigliato agli sviluppatori di script Python che hanno bisogno di eseguire il debug del loro codice Python, ad esempio degli id degli elementi o delle geometrie.
- `QgsLogger` è per i messaggi per il debugging *interno* di QGIS e per gli sviluppatori (ad esempio, si sospetta che qualcosa sia stato attivato da un codice non funzionante). I messaggi sono visibili solo con le versioni per sviluppatori di QGIS.

Gli esempi dei diversi tipi di registrazione sono illustrati nelle sezioni seguenti.

Avvertimento: L'uso dell'istruzione ``print"" di Python non è sicuro in qualsiasi codice che possa essere multithread e **rallenta enormemente l'algoritmo**. Ciò include **funzioni di espressione**, **visualizzazione**, **livelli di simboli** e **algoritmi di elaborazione** (tra gli altri). In questi casi si dovrebbe sempre usare il modulo **logging** di python o classi thread safe (`QgsLogger` o `QgsMessageLog`).

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!
```

Nota: Puoi vedere l'output di `QgsMessageLog` nel log_message_panel.

13.3.2 Il modulo di logging integrato in python

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

Il metodo basicConfig configura la configurazione di base del logging. Nel codice precedente vengono definiti il nome del file, il livello di registrazione e il formato. Il nome del file si riferisce a dove scrivere il file di log, il livello di log definisce i livelli di output e il formato definisce il formato di output di ciascun messaggio.

```
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
```

Se vuoi cancellare il file di log ogni volta che esegui lo script, puoi fare qualcosa di simile:

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

Ulteriori risorse su come utilizzare la funzione di registrazione in python sono disponibili all'indirizzo:

- 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

Avvertimento: Si noti che se non si esegue la registrazione su un file impostando un nome di file, la registrazione può essere multithread, il che rallenta notevolmente l'output.

CAPITOLO 14

Infrastruttura di autenticazione

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

```
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 Introduzione

Riferimenti per l'utente all'infrastruttura di autenticazione possono essere letti nel Manuale dell'utente nel paragrafo authentication_overview.

Questo capitolo descrive le migliori pratiche per utilizzare il sistema di autenticazione dal punto di vista dello sviluppatore.

Il sistema di autenticazione è ampiamente utilizzato in QGIS Desktop dai fornitori di dati ogni volta che sono richieste le credenziali per accedere a una particolare risorsa, ad esempio quando un layer stabilisce una connessione a un database Postgres.

Nella libreria QGIS gui sono presenti anche alcuni widget che gli sviluppatori di plugin possono utilizzare per integrare facilmente l'infrastruttura di autenticazione nel proprio codice:

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

Un buon riferimento per il codice può essere letto nell" infrastruttura di autenticazione tests code.

Avvertimento: A causa dei vincoli di sicurezza presi in considerazione durante la progettazione dell'infrastruttura di autenticazione, solo un sottoinsieme selezionato dei metodi interni è aperto a Python.

14.2 Glossario

Ecco qualche definizione degli oggetti più comuni trattati in questo capitolo.

Master Password
: Password che permette l'accesso e la decriptazione delle credenziali memorizzate nel DB di autenticazione di QGIS.

Database di autenticazione
: Una *Master Password* criptato sqlite db `qgis-auth.db` in cui *Authentication Configuration* sono memorizzati. ad esempio utente/password, certificati e chiavi personali, autorità di certificazione.

Autenticazione DB
: *Authentication Database*

Configurazione autenticazione
: Un insieme di dati di autenticazione che dipende da *Authentication Method*. Ad esempio, il metodo di autenticazione di base memorizza la coppia utente/password.

Authentication Config
: *Authentication Configuration*

Authentication Method
: Un metodo specifico utilizzato per ottenere l'autenticazione. Ogni metodo ha un proprio protocollo utilizzato per ottenere il livello di autenticazione. Ogni metodo è implementato come libreria condivisa caricata dinamicamente durante l'avvio dell'infrastruttura di autenticazione di QGIS.

14.3 QgsAuthManager il punto di ingresso

Il singleton `QgsAuthManager` è il punto di ingresso per utilizzare le credenziali memorizzate nel *Authentication DB* criptato di QGIS, ovvero il file `qgis-auth.db` sotto la cartella attiva user profile.

Questa classe si occupa dell'interazione con l'utente: chiedendo di impostare una master password o utilizzandola in modo trasparente per accedere alle informazioni crittografate memorizzate.

14.3.1 Avviare il manager e impostare la master password

Il seguente frammento di codice fornisce un esempio di impostazione della master password per aprire l'accesso alle impostazioni di autenticazione. I commenti al codice sono importanti per la comprensione del codice.

```
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 Popolare authdb con una nuova voce di configurazione dell'autenticazione

Ogni credenziale memorizzata è un'istanza *Authentication Configuration* della classe `QgsAuthMethodConfig` a cui si accede utilizzando una stringa univoca come la seguente:

```
authcfg = 'fm1s770'
```

Questa stringa viene generata automaticamente quando si crea una voce utilizzando l'API o la GUI di QGIS, ma potrebbe essere utile impostarla manualmente su un valore noto nel caso in cui la configurazione debba essere condivisa (con credenziali diverse) tra più utenti all'interno di un'organizzazione.

`QgsAuthMethodConfig` è la classe base per qualsiasi *Authentication Method*. Ogni metodo di autenticazione imposta una mappa hash di configurazione in cui saranno memorizzate le informazioni di autenticazione. Di seguito un utile frammento di codice per memorizzare le credenziali del percorso PKI per un ipotetico utente alice:

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

(continues on next page)

(continua dalla pagina precedente)

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

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

Metodi di autenticazione disponibili

Le librerie *Authentication Method* sono caricate dinamicamente durante l'avvio del gestore di autenticazione. I metodi di autenticazione disponibili sono:

1. `Basic` Autenticazione utente e password
2. `EsriToken` Autenticazione basata su token ESRI
3. `Identity-Cert` Identificazione certificato autenticazione
4. `OAuth2` autenticazione OAuth2
5. `PKI-Paths` Autenticazione percorsi PKI
6. `PKI-PKCS#12` Autenticazione PKI PKCS#12

Popolare le Autorità

```
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()
```

Gestire i pacchetti PKI con QgsPkiBundle

Una classe di comodo per confezionare pacchetti PKI composti da SslCert, SslKey e catena di CA è la classe `QgsPkiBundle`. Di seguito uno frammento di codice per ottenere una password protetta:

```
# 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()
```

Fare riferimento alla documentazione della classe `QgsPkiBundle` per estrarre cert/key/CA dal pacchetto.

14.3.3 Rimuovere una voce da authdb

Possiamo rimuovere una voce da *Authentication Database* usando il suo identificatore `authcfg` con il seguente frammento di codice:

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

14.3.4 Lasciare l'espansione di authcfg a QgsAuthManager

Il modo migliore per utilizzare una *Authentication Config* memorizzata nel *Authentication DB* è riferirla con l'identificatore unico `authcfg`. Espandere significa convertire l'identificatore in un insieme completo di credenziali. La pratica migliore per utilizzare le Authentication Config` memorizzate è quella di lasciarle gestire automaticamente dal gestore dell'autenticazione. L'uso comune di una configurazione memorizzata è quello di connettersi a un servizio abilitato all'autenticazione, come un WMS o un WFS, o a una connessione DB.

Nota: Tenere presente che non tutti i fornitori di dati QGIS sono integrati con l'infrastruttura di autenticazione. Ogni metodo di autenticazione, derivato dalla classe base `QgsAuthMethod`, supporta un diverso insieme di provider. Ad esempio, il metodo `certIdentity()` supporta il seguente elenco di provider:

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

Esempio di output:

```
['ows', 'wfs', 'wcs', 'wms', 'postgres']
```

Ad esempio, per accedere a un servizio WMS utilizzando le credenziali memorizzate identificate con `authcfg = 'fm1s770'`, è sufficiente utilizzare `authcfg` nell'URL dell'origine dati, come nel seguente frammento di codice:

```
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')
```

Nel caso di cui sopra, il provider `wms` avrà cura di espandere il parametro URI `authcfg` con le credenziali appena prima di impostare la connessione HTTP.

Avvertimento: Lo sviluppatore dovrebbe lasciare l'espansione di ``authcfg"" alla `QgsAuthManager`, in questo modo sarà sicuro che l'espansione non venga fatta troppo presto.

Di solito una stringa URI, costruita usando la classe `QgsDataSourceURI`, viene usata per impostare un'origine dati nel modo seguente:

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

Nota: Il parametro `False` è importante per evitare l'espansione completa dell'URI dell'id `authcfg` presente nell'URI.

Esempi di PKI con altri fornitori di dati

Altri esempi possono essere letti direttamente nei test di QGIS a monte, come in test_authmanager_pki_ows o test_authmanager_pki_postgres.

14.4 Adattare i plugin per utilizzare l'infrastruttura di autenticazione

Molti plugin di terze parti utilizzano httplib2 o altre librerie di rete Python per gestire le connessioni HTTP, invece di integrarsi con `QgsNetworkAccessManager` e la relativa integrazione dell'infrastruttura di autenticazione.

Per facilitare questa integrazione è stata creata una funzione helper in Python chiamata `NetworkAccessManager`. Il suo codice si trova here.

Questa helper classe può essere utilizzata come nel seguente frammento di codice:

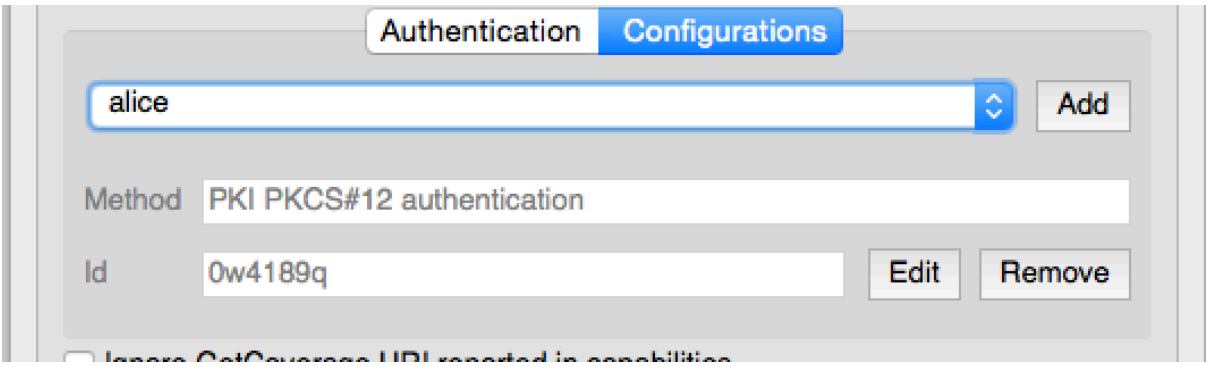
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 GUI di autenticazione

In questo paragrafo sono elencate le GUI disponibili utili per integrare l'infrastruttura di autenticazione in interfacce personalizzate.

14.5.1 GUI per selezionare le credenziali

Se è necessario selezionare una *Authentication Configuration* dall'insieme memorizzato nel *Authentication DB*, è disponibile nella classe GUI `QgsAuthConfigSelect`.



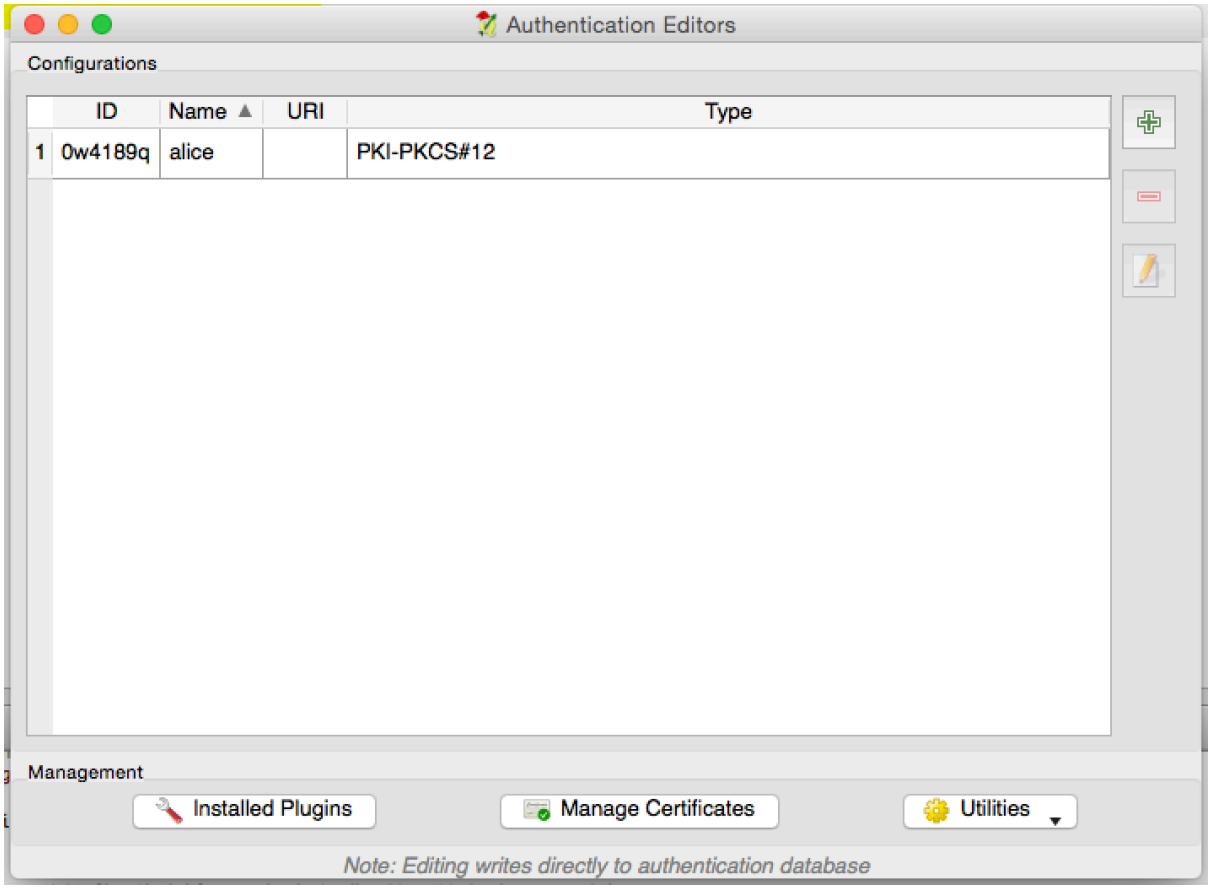
e può essere utilizzato come nel seguente frammento di codice:

```
# 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" )
```

L'esempio precedente è tratto dal sorgente QGIS code. Il secondo parametro del costruttore GUI si riferisce al tipo di fornitore di dati. Il parametro viene utilizzato per limitare i *Authentication Method* compatibili con il provider specificato.

14.5.2 Editor autenticazione GUI

L'interfaccia grafica completa utilizzata per gestire le credenziali, le autorità e per accedere alle utilità di autenticazione è gestita dalla classe `QgsAuthEditorWidgets`.



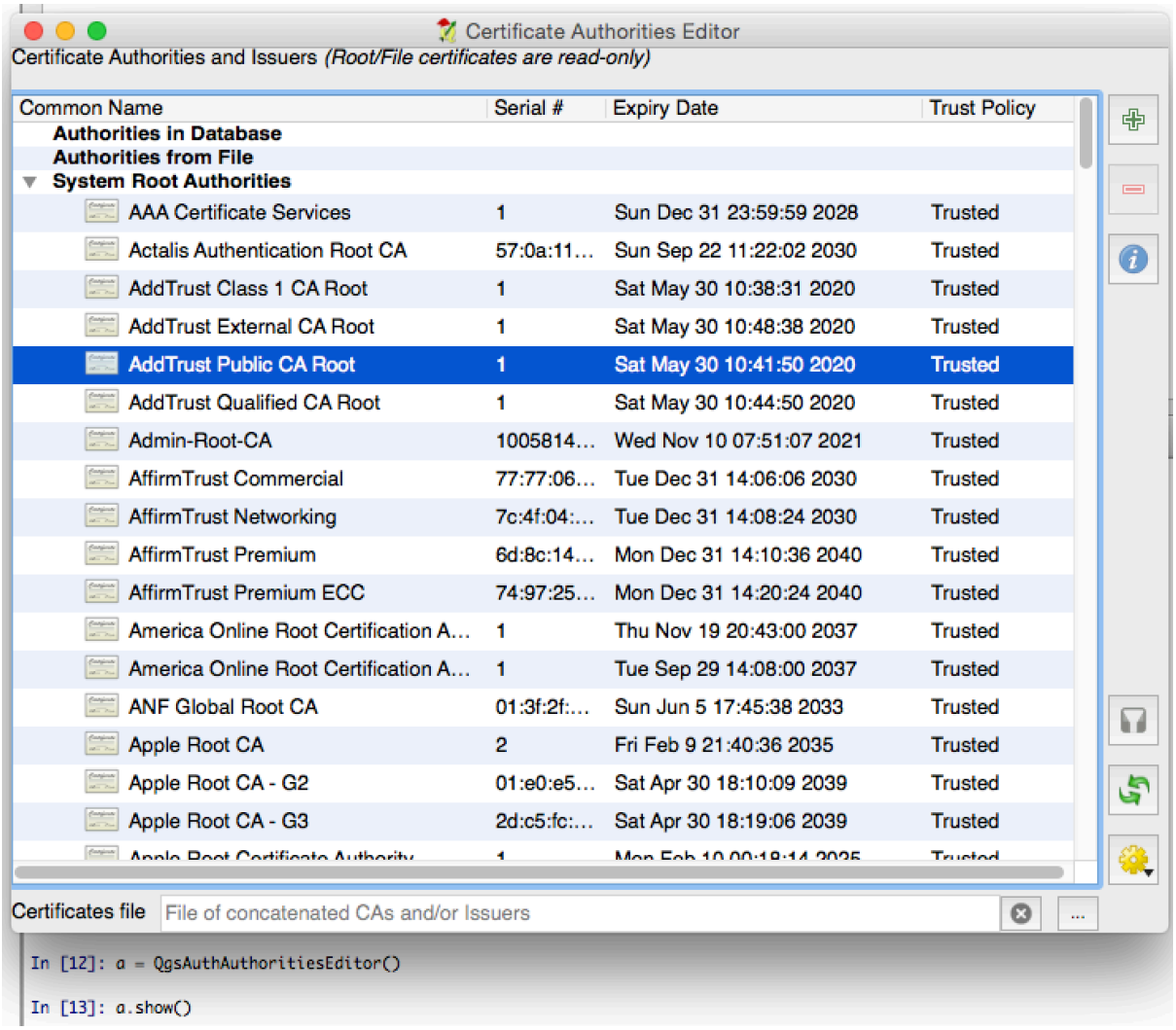
e può essere utilizzato come nel seguente frammento di codice:

```
# 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()
```

Un esempio integrato si trova nel relativo test.

14.5.3 Autorità Editor GUI

L'interfaccia grafica utilizzata per gestire solo le autorità è gestita dalla classe `QgsAuthAuthoritiesEditor`.



e può essere utilizzato come nel seguente frammento di codice:

```
# 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()
```

CAPITOLO 15

Task - fare un lavoro pesante in background

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

15.1 Introduzione

L'elaborazione in background tramite thread è un modo per mantenere un'interfaccia utente reattiva quando è in corso un'elaborazione pesante. I task possono essere utilizzati per ottenere il threading in QGIS.

Un task (`QgsTask`) è un contenitore per il codice da eseguire in background e il task manager (`QgsTaskManager`) è usato per controllare l'esecuzione dei task. Queste classi semplificano l'elaborazione in background in QGIS, fornendo meccanismi per la segnalazione, l'avanzamento e l'accesso allo stato dei processi in background. I task possono essere raggruppati utilizzando i sottotask.

Normalmente viene utilizzato il task manager globale (trovato con `QgsApplication.taskManager()`). Ciò significa che i tuoi task potrebbero non essere gli unici controllati dal task manager.

Esistono diversi modi per creare un task QGIS:

- Creare una tua attività personalizzata estendendo `QgsTask`

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

- Creare un task da una funzione

```
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)
```

- Creare un task da un algoritmo di processing

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

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

Avvertimento: Qualsiasi attività in background (indipendentemente da come è stata creata) non deve MAI utilizzare alcun QObject che vive sul thread principale, come ad esempio accedere a QgsVectorLayer, QgsProject o eseguire operazioni basate sull'interfaccia grafica, come la creazione di nuovi widget o l'interazione con i widget esistenti. I widget Qt devono essere accessibili o modificati solo dal thread principale. I dati utilizzati in un'attività devono essere copiati prima dell'avvio dell'attività stessa. Il tentativo di utilizzarli da thread in background provocherà un arresto anomalo.

Inoltre, assicurati sempre che `context` e `feedback` rimangano in vita almeno quanto le attività che li utilizzano. QGIS si arresta in modo anomalo se, al completamento di un'attività, *QgsTaskManager* non riesce ad accedere al *contesto* e al *feedback* rispetto ai quali l'attività è stata pianificata.

Nota: È uno schema comune chiamare `setProject()` poco dopo aver chiamato `QgsProcessingContext`. Questo permette al task e alla sua funzione di callback di utilizzare la maggior parte delle impostazioni del progetto. Questo è particolarmente utile quando si lavora con i livelli spaziali nella funzione di callback.

Le dipendenze tra attività possono essere descritte utilizzando la funzione `addSubTask()` di `QgsTask`. Quando viene dichiarata una dipendenza, il task manager determina automaticamente il modo in cui queste dipendenze saranno eseguite. Ove possibile, le dipendenze saranno eseguite in parallelo, in modo da soddisfarle il più rapidamente possibile. Se un task da cui dipende un altro task viene annullato, anche il task dipendente verrà annullato. Le dipendenze circolari possono rendere possibili i deadlock, quindi bisogna fare attenzione.

Se un'attività dipende dalla disponibilità di un layer, questo può essere dichiarato utilizzando la funzione `setDependentLayers()` della `QgsTask`. Se un layer da cui dipende un task non è disponibile, il task viene annullato.

Una volta creato, il task può essere pianificato per l'esecuzione utilizzando la funzione `addTask()` del task manager. L'aggiunta di un task al gestore ne trasferisce automaticamente la proprietà al gestore stesso, che provvederà a ripulire e cancellare i task dopo la loro esecuzione. La programmazione dei task è influenzata dalla priorità del task, impostata in `addTask()`.

Lo stato delle attività può essere monitorato utilizzando i servizi e le funzioni `QgsTask` e `QgsTaskManager`.

15.2 Esempi

15.2.1 Estensione di QgsTask

In questo esempio `RandomIntegerSumTask` estende `QgsTask` e genererà 100 numeri interi casuali tra 0 e 500 durante un periodo di tempo specificato. Se il numero casuale è 42, il task viene interrotto e viene sollevata un'eccezione. Diverse istanze di `RandomIntegerSumTask` (con sottoattività) vengono generate e aggiunte al task manager, dimostrando due tipi di dipendenze.

```
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)

(continua dalla pagina precedente)

```
        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 Task da funzione

Crea un task da una funzione (`doSomething` in questo esempio). Il primo parametro della funzione conterrà la `QgsTask` per la funzione. Un parametro importante (con nome) è `on_finished`, che specifica una funzione che sarà chiamata quando il task è stato completato. La funzione `doSomething` in questo esempio ha un parametro aggiuntivo chiamato `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)

(continua dalla pagina precedente)

```
        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 Task da un algoritmo di processing

Crea un task che utilizza l'algoritmo qgis:randompointsinextent per generare 50000 punti casuali all'interno di un'estensione specificata. Il risultato viene aggiunto al progetto in modo sicuro.

```
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)

(continua dalla pagina precedente)

```
    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)
```

Vedi anche: https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/.

CAPITOLO 16

Sviluppo di plugin Python

16.1 Struttura Plugin Python

I passi principali per la creazione di un plugin sono:

1. *Idea*: Avere un'idea di ciò che si vuole fare con il nuovo plugin QGIS.
2. *Impostazione*: *Create the files for your plugin*. A seconda del tipo di plugin, alcuni sono obbligatori mentre altri sono opzionali
3. *Sviluppo*: *Write the code* nei file appropriati
4. *Documentazione*: *Write the plugin documentation*
5. Opzionalmente: *Tradurre*: *Translate your plugin* in diverse lingue
6. *Test*: *Ricarica il tuo plugin* per verificare che tutto è a posto
7. *Pubblica*: pubblica il tuo plugin nel repository di QGIS ocrea il tuo repository personale come «magazzino» di «strumenti GIS» personali.

16.1.1 Come iniziare

Prima di iniziare a scrivere un nuovo plugin, dai un'occhiata al *Repository ufficiale dei plugin Python*. Il codice sorgente dei plugin esistenti può aiutare a imparare di più sulla programmazione. Potresti anche scoprire che esiste già un plugin simile e che potresti essere in grado di estenderlo o almeno di basarti su di esso per sviluppare il tuo.

Imposta la struttura dei file del plugin

Per iniziare con un nuovo plugin, dobbiamo impostare i file necessari per il plugin.

Esistono due risorse relative a modelli di plugin che possono aiutarti a iniziare:

- A scopo didattico o quando si desidera un approccio minimalista, il minimal plugin template fornisce i file di base (scheletro) necessari per creare un plugin QGIS Python valido.
- Per un modello di plugin più completo, il Plugin Builder può creare modelli per diversi tipi di plugin, incluse funzionalità come la localizzazione (traduzione) e il testing.

Una tipica cartella contenente plugin comprende i seguenti file:

- `metadata.txt` - *obbligatorio* - Contiene informazioni generali, versione, nome e alcuni altri metadati utilizzati dal sito web e dall'infrastruttura del plugin.
- `__init__.py` - *obbligatorio* - Il punto di lancio del plugin. Deve avere il metodo `classFactory()` e può avere qualsiasi altro codice di inizializzazione.
- `mainPlugin.py` - *codice di base* - Il codice di base del plugin. Contiene tutte le informazioni sulle azioni del plugin e il codice di base.
- `form.ui` - *per i plugin con GUI personalizzata* - L'interfaccia grafica creata da Qt Designer.
- `form.py` - *GUI compilata* - La traduzione del form.ui descritto sopra in Python.
- `resources.qrc` - *opzionale* - Un documento .xml creato da Qt Designer. Contiene i percorsi relativi alle risorse utilizzate nei moduli della GUI.
- `resources.py` - *risorse compilate, opzionale* - La traduzione in Python del file .qrc descritto sopra.
- `LICENSE` - *required* if plugin is to be published or updated in the QGIS Plugins Directory, otherwise *optional*. File should be a plain text file with no file extension in the filename.

Avvertimento: Se tu vuoi caricare il plugin sul *Repository ufficiale dei plugin Python* devi controllare che il tuo plugin segua alcune regole aggiuntive, obbligatorie per la *Validazione* del plugin.

16.1.2 Scrivere codice per i plugin

La sezione seguente mostra il contenuto da aggiungere in ciascuno dei file introdotti in precedenza.

metadata.txt

Per prima cosa, il Plugin Manager deve recuperare alcune informazioni di base sul plugin, come il nome, la descrizione ecc. Queste informazioni sono memorizzate in `metadata.txt`.

Nota: Tutti i metadati devono essere in codifica UTF-8.

Nome Metadati	Richies	Note
nome	Vero	Una breve stringa contenente il nome del plugin
qgisMinimum`	Vero	notazione della versione minima di QGIS
qgisMaximum	Falso	notazione della versione massima
Descrizione	Vero	breve testo che descrive il plugin, non in HTML
su	Vero	testo più lungo che descrive il plugin nei dettagli, non è permesso HTML
versione	Vero	corta stringa con la notazione della versione
autore	Vero	nome dell'autore
email	Vero	email dell'autore, mostrato solo sul sito web per l'accesso degli utenti, ma visibile nel gestore plugin solo dopo l'installazione del plugin
cambiamenti	Falso	stringa, può essere multilinea, HTML non permesso
sperimentale	Falso	contrassegno, `Vero` o `Falso` - `Vero` se la versione è sperimentale
deprecato	Falso	contrassegno, `Vero` o `Falso`, si applica a tutto il plugin e non solo alla versione caricata
etichette	Falso	Elenco separato da virgola, gli spazi sono ammessi all'interno delle singole etichette
homepage	Falso	un URL valido per la pagina iniziale del tuo plugin
repository	Vero	un URL valido per il repository del codice sorgente
tracker	Falso	un URL valido per ticket e bug
icona	Falso	a file name or a relative path (relative to the base folder of the plugin's compressed package) of a web friendly image (PNG, JPEG)
categoria	Falso	una tra `Raster`, `Vettore`, `Database`, `Mesh` e `Web`
plugin_depend	Falso	Elenco separato da virgole, simile a un PIP, di altri plugin da installare; utilizza i nomi dei plugin provenienti dal campo del nome dei rispettivi metadati.
server	Falso	contrassegno, `Vero` o `Falso`, determina se il plugin ha una interfaccia server
hasProcessingl	Falso	contrassegno, `Vero` o `Falso`, determina se il plugin prevede algoritmi processing

Per impostazione predefinita, i plugin sono inseriti nel menu *Plugin* (vedremo nella prossima sezione come aggiungere una voce di menu per il tuo plugin), ma possono anche essere inseriti nei menu *Raster*, *Vettore*, *Database*, *Mesh* e *Web*.

Una corrispondente voce di metadati: «categoria» esiste per specificare come può essere classificato il plugin. Questa voce dei metadati viene utilizzata per gli utenti e dice loro dove (in quale menu) è possibile trovare il plugin. I valori consentiti per «categoria» sono: vettoriali, raster, database o web. Ad esempio, se il tuo plugin sarà disponibile dal menu raster, aggiungilo a metadata.txt

```
category=Raster
```

Nota: Se`qgisMaximumVersion` è vuoto, sarà automaticamente impostata l'ultima versione più *.99* quando caricato su *Repository ufficiale dei plugin Python*.

Un esempio di metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=3.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
```

(continues on next page)

(continua dalla pagina precedente)

```
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

Questo file è richiesto dal sistema di importazione di Python. Inoltre, QGIS richiede che questo file contenga una funzione `classFactory()`, che viene chiamata quando il plugin viene caricato in QGIS. Riceve un riferimento all'istanza di `QgisInterface` e deve restituire un oggetto della classe del tuo plugin dal file `mainplugin.py` — nel nostro caso si chiama `TestPlugin` (vedi sotto). Questo è come dovrebbe apparire `__init__.py`

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

# any other initialisation needed
```

mainPlugin.py

Qui è dove avviene e come appare la magia: (e.g. `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!")
```

Le sole funzioni plugin che devono esistere nel file sorgente del plugin principale (e.g. `mainPlugin.py`) sono:

- `__init__` che da accesso all'interfaccia QGIS
- `initGui()` chiamata se il plugin è caricato
- `unload()` cchiamata quando il plugin è scaricato

Nell'esempio precedente, viene utilizzato `addPluginToMenu()`. Questo aggiungerà l'azione del menu corrispondente al menu *Plugins*. Esistono metodi alternativi per aggiungere l'azione a un menu diverso. Ecco un elenco di quei metodi:

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

- `addPluginToDatabaseMenu()`
- `addPluginToWebMenu()`

Tutti hanno la stessa sintassi del metodo `addPluginToMenu()`.

Si consiglia di aggiungere il tuo menu plugin con uno di questi metodi predefiniti per mantenere la coerenza nel modo in cui sono organizzati i plugin nel menu. Tuttavia, è possibile aggiungere il tuo gruppo di menu personalizzato direttamente alla barra dei menu, come dimostra il prossimo esempio:

```
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()
```

Non dimenticare di impostare `QAction` e `QMenu` `objectName` con un nome specifico per il tuo plugin in modo che possa essere personalizzato.

Anche se le azioni di aiuto e di informazione possono essere aggiunte al menu personalizzato, un posto conveniente per renderle disponibili è il menu principale di QGIS *Guida* ► *Plugin*. Per farlo si utilizza il metodo `pluginHelpMenu()`.

```
def initGui(self):

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

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

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

def unload(self):

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

Se lavori su un plugin reale è saggio scrivere il plugin in un'altra directory (funzionante) e creare un makefile che genererà file UI + file risorse e installa il plugin nella tuainstallazione QGIS.

16.1.3 Documentare plugin

Puoi scrivere la documentazione per il plugin come file HTML. Il modulo `qgis.utils` fornisce una funzione, `showPluginHelp()` che aprirà il browser del file della Guida, allo stesso modo dell'aiuto di QGIS.

La funzione:func:*showPluginHelp* cerca i file di aiuto nella stessa cartella del modulo di chiamata. Cercherà, a turno, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` e `index.html`. Qui `ll_cc` is the QGIS. Ciò consente a più traduzioni della documentazione da includere nel plugin.

La funzione `showPluginHelp()` può anche accettare i parametri packageName, che identifica uno specifico plugin per il quale verrà visualizzata la guida, filename, che può sostituire «index» nei nomi dei file da ricercare, e section, che è il nome di un anchor tag html nel documento su cui verrà posizionato il browser.

16.1.4 Tradurre plugin

Con pochi passaggi puoi impostare l'ambiente per la localizzazione del plugin, in modo che, a seconda delle impostazioni locali del computer, il plugin venga caricato in lingue diverse.

Requisiti software

The easiest way to create and manage all the translation files is to install Qt Linguist. In a Debian-based GNU/Linux environment you can install it typing:

```
sudo apt install qttools5-dev-tools
```

File e cartella

Quando crei il plugin troverai la cartella `i18n` all'interno della cartella principale dei plugin.

Tutti i file di traduzione devono trovarsi in questa cartella.

.pro file

Per prima cosa devi creare un file ``.pro"", cioè un file *progetto* che può essere gestito da **Qt Linguist**.

In questo file ``.pro"" devi specificare tutti i file e i moduli che vuoi tradurre. Questo file viene usato per impostare i file di localizzazione e le variabili. Un possibile file di progetto, che corrisponde alla struttura del nostro *example plugin*:

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

Il tuo plugin potrebbe avere una struttura più complessa ed essere distribuito in diversi file. In questo caso, tieni presente che `pylupdate5`, il programma che si usa per leggere il file `.pro` e aggiornare la stringa traducibile, non espande i caratteri jolly, quindi devi inserire ogni file esplicitamente nel file `.pro`. Il file di progetto potrebbe quindi avere un aspetto simile a questo:

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

Inoltre, il file `tuo_plugin.py` è il file che *richiama* tutti i menu e i sottomenu del tuo plugin nella barra degli strumenti di QGIS e tu vuoi tradurli tutti.

Infine, con la variabile *TRANSLATIONS* puoi specificare le lingue di traduzione volute.

Avvertimento: Assicurati di denominare il file `ts` come `tuo_plugin_` + `language` + `.ts` altrimenti il caricamento della lingua fallirà! Utilizza la scorciatoia di 2 lettere per la lingua (**it** per l'italiano, **de** per il tedesco, ecc…)

.ts file

Una volta che hai creato il file `.pro`, sei pronto a generare il(i) file `.ts` per la lingua(e) del plugin.

Apri un terminale, vai nella cartella `tuo_plugin/i18n` e digita:

```
pylupdate5 your_plugin.pro
```

dovresti vedere il(i) file `tuo_plugin_language.ts`.

Apri il file ``.ts"" con **Qt Linguist** e inizia a tradurre.

.qm file

Quando hai finito di tradurre il plugin (se alcune stringhe non sono state completate, verrà usata la lingua di partenza per quelle stringhe), devi creare il file `.qm` (il file `.ts` compilato che verrà usato da QGIS).

Devi solo aprire un terminale con un cd nella cartella `tuo_plugin/i18n` e digitare:

```
lrelease your_plugin.ts
```

ora, nella cartella `i18n` vedrai il(i) file `tuo_plugin.qm`.

Tradurre con Makefile

In alternativa, puoi usare il makefile per estrarre i messaggi dal codice python e dalle finestre di dialogo Qt, se hai creato il plugin con Plugin Builder. All'inizio del Makefile c'è una variabile LOCALES:

```
LOCALES = en
```

Aggiungi l'abbreviazione della lingua a questa variabile, ad esempio per la lingua ungherese:

```
LOCALES = en hu
```

Ora puoi generare o aggiornare il file `hu.ts` (e anche `en.ts`) dai sorgenti da:

```
make transup
```

In questo modo, hai il file `.ts` aggiornato per tutte le lingue impostate nella variabile LOCALES. Utilizza **Qt Linguist** per tradurre i messaggi del programma. Al termine della traduzione, i file `.qm` possono essere creati da transcompile:

```
make transcompile
```

Devi installare i file ``.ts"" con il tuo plugin.

Caricare il plugin

Per vedere la traduzione del plugin, apri QGIS, cambia la lingua (*Impostazioni* ► *Opzioni* ► *Generale*) e riavvia QGIS.

Dovresti vedere il tuo plugin nella lingua corretta.

> **Avvertimento:** Sei modifichi qualcosa nel plugin (nuove UI, nuovi menu, ecc.) devi **generare nuovamente** la versione aggiornata di entrambi i file `.ts` e `.qm`, quindi eseguire nuovamente il comando di cui sopra.

16.1.5 Condividere il tuo plugin

QGIS ospita centinaia di plugin nel repository dei plugin. Considera di condividere il tuo! Estenderà le possibilità di QGIS e le persone potranno imparare dal tuo codice. Tutti i plugin ospitati possono essere trovati e installati dall'interno di QGIS con il Plugin Manager.

Informazioni e requisiti sono disponibili qui: plugins.qgis.org.

16.1.6 Suggerimenti e trucchi

Plugin Reloader

Durante lo sviluppo del tuo plugin sarà spesso necessario ricaricarlo in QGIS per testarlo. Questo è molto semplice utilizzando il plugin **Plugin Reloader**. È possibile trovarlo con il comando Plugin Manager.

Automatizzare la pacchettizzazione, il rilascio e la traduzione con qgis-plugin-ci

qgis-plugin-ci provides a command line interface to perform automated packaging and deployment for QGIS plugins on your computer, or using continuous integration like GitHub workflows or Gitlab-CI as well as Transifex for translation.

Permette di rilasciare, tradurre, pubblicare o generare un file XML di repository di plugin tramite CLI o in azioni CI.

Accessere ai plugin

Puoi accedere a tutte le classi dei plugin installati dall'interno di QGIS usando python, il che può essere utile a fini di debug.

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

Messaggi di Log

I plugin hanno una propria scheda all'interno del log_message_panel.

File delle risorse

Alcuni plugin utilizzano file di risorse, ad esempio `resources.qrc` che definiscono risorse per la GUI, come le icone:

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

È bene usare un prefisso che non si scontra con altri plugin o parti di QGIS, altrimenti potresti ottenere risorse che non vuoi. Ora hai solo bisogno di generare un file Python che conterrà le risorse. È fatto con il comando **`pyrcc5`**:

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

Nota: In ambienti Windows, il tentativo di eseguire **`pyrcc5`** dal prompt dei comandi o PowerShell probabilmente comporterà l'errore «Windows non può accedere al dispositivo, al percorso o al file specificato […].». La soluzione più semplice è probabilmente quella di utilizzare la shell OSGeo4W ma se puoi modificare la variabile ambiente del percorso o specifichi il percorso verso l'eseguibile dovresti essere in grado di trovarela in `<Your QGIS Install Directory>\bin\pyrcc5.exe`.

16.2 Blocchi di codice

Suggerimento: I frammenti di codice di questa pagina necessitano delle seguenti importazioni se sei è al di fuori della console pyqgis:

```
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
```

Questa sezione contiene frammenti di codice per facilitare lo sviluppo dei plugin.

16.2.1 Come richiamare un metodo con una scorciatoia da tastiera

Nel plug-in aggiungere a `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)
```

A `unload()` aggiungere

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

Il metodo che viene richiamato quando si preme CTRL+I

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

È anche possibile consentire agli utenti di personalizzare le scorciatoie da tastiera per le azioni fornite. Questo si ottiene aggiungendo:

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

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

16.2.2 Come riutilizzare le icone di QGIS

Poiché sono ben conosciute e trasmettono un messaggio chiaro agli utenti, a volte potresti voler riutilizzare le icone di QGIS nel tuo plugin, invece di disegnarne e impostarne una nuova. Utilizza il metodo `getThemeIcon()`.

Ad esempio, per riutilizzare l'icona  mActionFileOpen.svg disponibile nel repository del codice QGIS:

```
# 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()` è un altro metodo per richiamare le icone di QGIS. Trovi esempi di richiami alle icone tematiche in QGIS embedded images - Cheatsheet.

16.2.3 Interfaccia per il plugin nella finestra di dialogo delle opzioni

Puoi aggiungere una scheda per le opzioni del plugin a *Impostazioni* ► *Opzioni*. Questa soluzione è preferibile all'aggiunta di una voce specifica del menu principale per le opzioni del plugin, in quanto consente di mantenere tutte le impostazioni dell'applicazione QGIS e del plugin in un unico punto, facile da scoprire e da navigare per gli utenti.

Il seguente frammento di codice aggiungerà una nuova scheda vuota per le impostazioni del plugin, pronta per essere popolata con tutte le opzioni e le impostazioni specifiche del plugin. Puoi dividere le classi seguenti in file diversi. In questo esempio, aggiungiamo due classi al file principale `mainPlugin.py`.

```
class MyPluginOptionsFactory(QgsOptionsWidgetFactory):

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

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

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


class ConfigOptionsPage(QgsOptionsPageWidget):

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

Infine, aggiungiamo le importazioni e modifichiamo la funzione `__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)
```

Suggerimento: Aggiungere schede personalizzate alla finestra di dialogo delle proprietà dei layer

Puoi adottare una logica simile per aggiungere l'opzione personalizzata del plugin alla finestra di dialogo delle proprietà del layer, usando le classi `QgsMapLayerConfigWidgetFactory` e `QgsMapLayerConfigWidget`.

16.2.4 Incorporare widget personalizzati per i layer nell'albero dei layer

Oltre ai consueti elementi della simbologia dei layer visualizzati accanto o sotto la voce del layer nel pannello *Layer*, puoi aggiungere i tuoi widget, consentendo l'accesso rapido ad alcune azioni spesso utilizzate con un layer (impostazione di filtri, selezioni, stili, aggiornamento di un layer con un widget pulsante, creazione di un cursore temporale basato sul layer o semplicemente visualizzazione di informazioni extra sul layer in un'etichetta, o …). Questi cosiddetti **Widget incorporati nell'albero dei layer** sono resi disponibili per i singoli layer attraverso le proprietà *Legenda* del layer.

Il seguente frammento di codice crea un menu a tendina nella legenda che mostra gli stili di livello disponibili per il layer, consentendo di passare rapidamente tra i diversi stili di livello.

```
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)
```

Quindi, dalla scheda delle proprietà *Legenda* di un dato layer, trascina la `Scelta stile layer` da *Widget disponibili* alla *Widget utilizzati*` per abilitare il widget nell'albero del layer. I widget incorporati sono SEMPRE visualizzati in cima alle sottovoci dei nodi di livello associati.

Se vuoi utilizzare i widget dall'interno, ad esempio, di un plugin, puoi aggiungerli in questo modo:

```
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 Impostazioni IDE per scrittura e debug dei plugin

Anche se ogni programmatore ha il suo IDE/editor di testo preferito, ecco alcuni consigli per impostare gli IDE più diffusi per la scrittura e il debug dei plugin Python di QGIS.

16.3.1 Plugin utili per scrivere plugin Python

Alcuni plugin sono comodi per la scrittura di plugin Python. Da *Plugins* ► *Gestisci ed Installa Plugin…*, installa:

- *Plugin Reloader*: Consente di ricaricare un plugin e di apportare nuove modifiche senza riavviare QGIS.
- *First Aid*: Aggiungerà una console Python e un debugger locale per ispezionare le variabili quando viene generata un'eccezione da un plugin.

Avvertimento: Nonostante i nostri costanti sforzi, le informazioni al di là di questa linea potrebbero non essere aggiornate per QGIS 3.

16.3.2 Una nota sulla configurazione di IDE su Linux e Windows

Su Linux, di solito è sufficiente aggiungere i percorsi delle librerie QGIS alla variabile d'ambiente `PYTHONPATH` dell'utente. Nella maggior parte delle distribuzioni, questo può essere fatto modificando `~/.bashrc` o `~/.bash-profile` con la seguente linea (testata su OpenSUSE Tumbleweed):

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

Salva il file e implementa le impostazioni di ambiente usando il seguente comando shell:

```
source ~/.bashrc
```

Su Windows, devi assicurarti di avere le stesse impostazioni di ambiente e di utilizzare le stesse librerie e lo stesso interprete di QGIS. Il modo più rapido per farlo è modificare il file batch di avvio di QGIS.

Se hai utilizzato il programma di installazione OSGeo4W, puoi trovarlo nella cartella `bin` dell'installazione di OSGeo4W. Cerca qualcosa come `C:\OSGeo4Wbinqgis-unstable.bat`.

16.3.3 Debug con Pyscripter IDE (Windows)

Per usare Pyscripter IDE, ecco cosa devi fare:

1. Fai una copia di `qgis-unstable.bat` e rinominalo `pyscripter.bat`.
2. Aprirlo in un editor. Rimuovi l'ultima riga, quella che avvia QGIS.
3. Aggiungi una linea che punti all'eseguibile di Pyscripter e aggiungi il parametro della linea di comando che imposta la versione di Python da utilizzare
4. Aggiungi anche il parametro che punta alla cartella in cui Pyscripter può trovare la dll di Python usata da QGIS, che si trova nella cartella bin dell'installazione di OSGeoW

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

5. Ora, quando fai doppio clic su questo file batch, viene avviato Pyscripter, con il percorso corretto.

Più popolare di Pyscripter, Eclipse è una scelta comune tra gli sviluppatori. Nella sezione che segue, spiegheremo come configurarlo per sviluppare e testare i plugin.

16.3.4 Debug con Eclipse e PyDev

Installazione

Per utilizzare Eclipse, assicurati di aver installato quanto segue

- Eclipse
- Aptana Studio 3 Plugin or PyDev
- QGIS 3.x
- Si potrebbe anche installare **Remote Debug**, un plugin di QGIS. Al momento è ancora sperimentale, per cui è necessario prima attivare la casella di controllo *Plugin sperimentali* in *Plugins* ► *Gestisci ed Installa Plugin…* ► *Impostazioni*.

Per preparare l'ambiente all'uso di Eclipse in Windows, occorre anche creare un file batch e usarlo per avviare Eclipse:

1. Individuare la cartella in cui risiede `qgis_core.dll`. Normalmente si tratta di `C:\OSGeo4W\apps\qgis\bin`, ma se hai compilato la tua applicazione QGIS questa si trova nella cartella di compilazione in `output/bin/RelWithDebInfo`.
2. Individua l'eseguibile `eclipse.exe`.
3. Crea il seguente script e utilizzalo per avviare eclipse durante lo sviluppo dei plugin QGIS.

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

Impostazione di Eclipse

1. In Eclipse, crea un nuovo progetto. Puoi selezionare *Progetto generale* e collegare i sorgenti reali in seguito, quindi non ha molta importanza dove si colloca questo progetto.

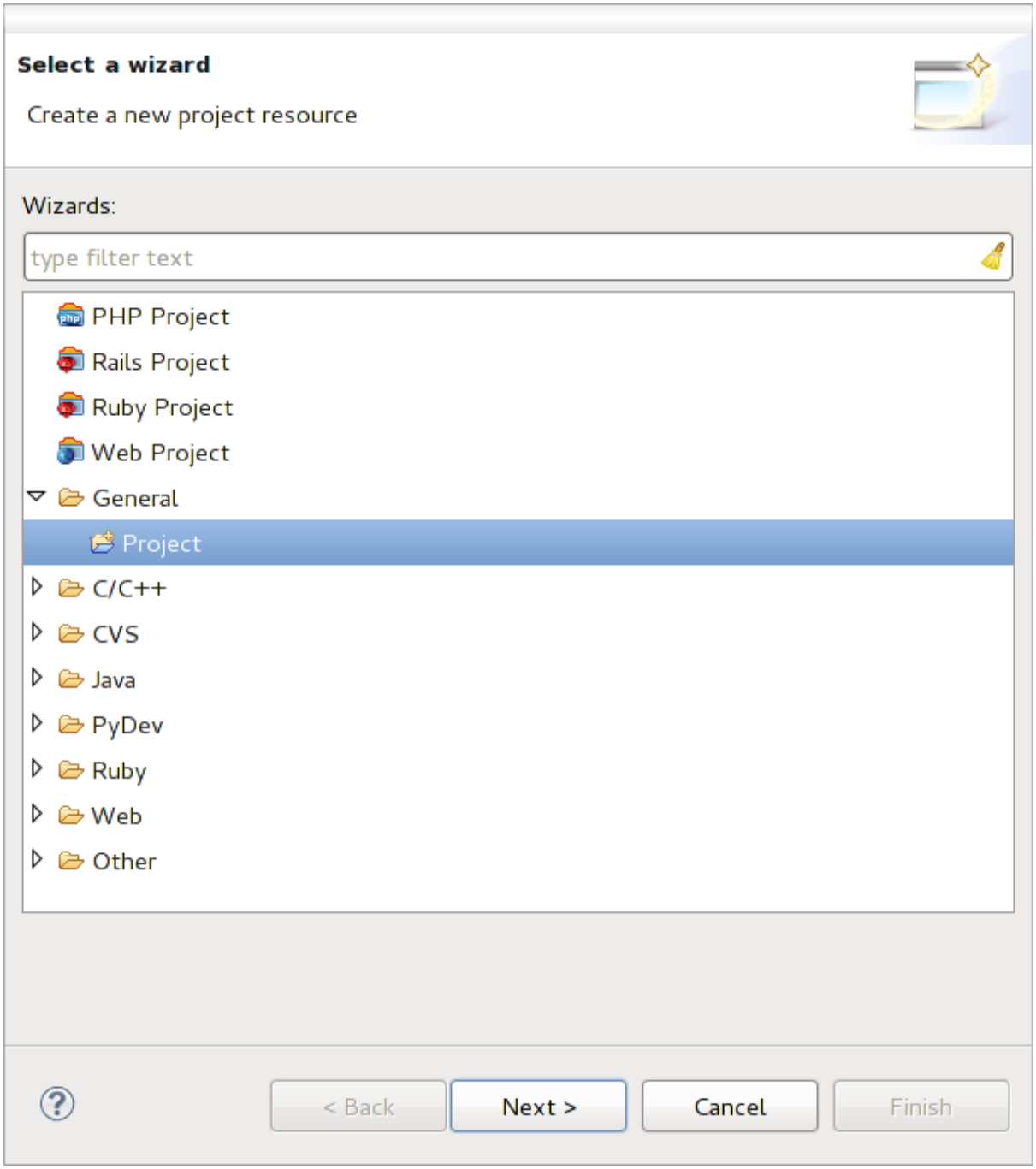


Fig. 16.1: Progetto Eclipse

2. Fai clic con il pulsante destro del mouse sul nuovo progetto e scegli *Nuovo* ► *Cartella*.
3. Fai clic su *Avanzate* e scegliete *Collegamento a un percorso alternativo (cartella collegata)*. Se disponi già di sorgenti di cui vuoi eseguire il debug, sceglile. In caso contrario, crea una cartella come già spiegato.

Ora nella vista *Project Explorer*, appare il tuo albero dei sorgenti e puoi iniziare a lavorare con il codice. Hai a disposizione l'evidenziazione della sintassi e tutti gli altri potenti strumenti dell'IDE.

Configurare il debugger

Per far funzionare il debugger:

1. Passa alla sezione Debug di Eclipse (*Window* ► *Open Perspective* ► *Other* ► *Debug*).
2. avvia il server di debug PyDev scegliendo *PyDev* ► *Start Debug Server*.
3. Eclipse è ora in attesa di una connessione da QGIS al suo server di debug e quando QGIS si connette al server di debug gli permetterà di controllare gli script python. È proprio per questo che abbiamo installato il plugin *Remote Debug*. Quindi avvia QGIS, se non l'hai già fatto, e fai clic sul simbolo del bug.

Ora puoi impostare un punto di interruzione e, non appena il codice lo raggiunge, l'esecuzione si interrompe e si può ispezionare lo stato attuale del plugin. (Il punto di interruzione è il punto verde nell'immagine sottostante; per impostarlo, fai doppio clic nello spazio bianco a sinistra della linea in cui vuoi che sia impostato il punto di interruzione).

```
 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: Punto di interruzione

Una cosa molto interessante che puoi utilizzare ora è la console di debug. Prima di procedere, assicurati che l'esecuzione si sia fermata a un punto di interruzione.

1. Apri la vista Console (*Window* ► *Show view*). Verrà mostrata la console *Debug Server* che non è molto interessante. Ma c'è un pulsante *Apri console* che consente di passare a una console di debug PyDev più interessante.
2. Fai clic sulla freccia accanto al pulsante *Apri console* e scegli *PyDev Console*. Si apre una finestra che chiede quale console vuoi avviare.
3. Scegli *Console di debug PyDev*. Nel caso in cui sia grigio e ti dica di avviare il debugger e di selezionare il frame valido, assicurati di aver collegato il debugger remoto e di trovarti in un punto di interruzione.

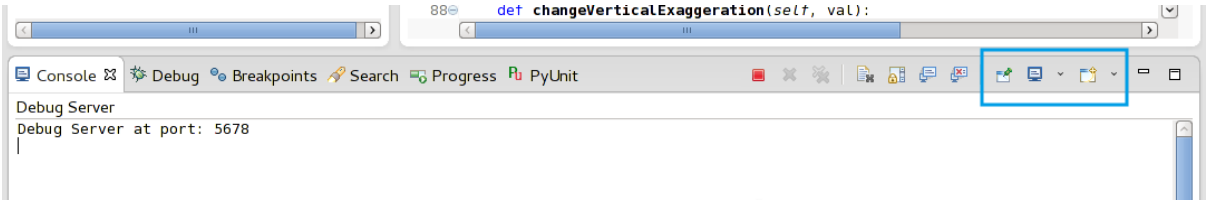


Fig. 16.3: Console di debug PyDev

Ora hai una console interattiva che consente di testare qualsiasi comando all'interno del contesto corrente. Puoi manipolare le variabili, effettuare chiamate API o qualsiasi altra cosa.

Suggerimento: Un po" fastidioso è che ogni volta che si immette un comando, la console passa di nuovo al server di debug. Per interrompere questo comportamento, puoi fare clic sul pulsante *Pin Console* quando sei nella pagina del server di debug e la console dovrebbe ricordare questa decisione almeno per la sessione di debug corrente.

Fare in modo che eclipse capisca l'API

Una funzione molto utile è quella di far conoscere a Eclipse l'API di QGIS. In questo modo può controllare che il codice non contenga errori di battitura. Ma non solo, Eclipse ti aiuta anche con il completamento automatico delle importazioni e delle chiamate API.

Per fare questo, Eclipse analizza i file delle librerie di QGIS e ottiene tutte le informazioni. L'unica cosa che devi fare è dire a Eclipse dove trovare le librerie.

1. Fai clic su **:menuselezione:`Window --> Preferences --> PyDev --> Interpreter --> Python`**.

   Nella parte superiore della finestra vedrai l'interprete python configurato (al momento python2.7 per QGIS) e alcune schede nella parte inferiore. Le schede interessanti per noi sono *Libraries* e *Forced Builtins*.

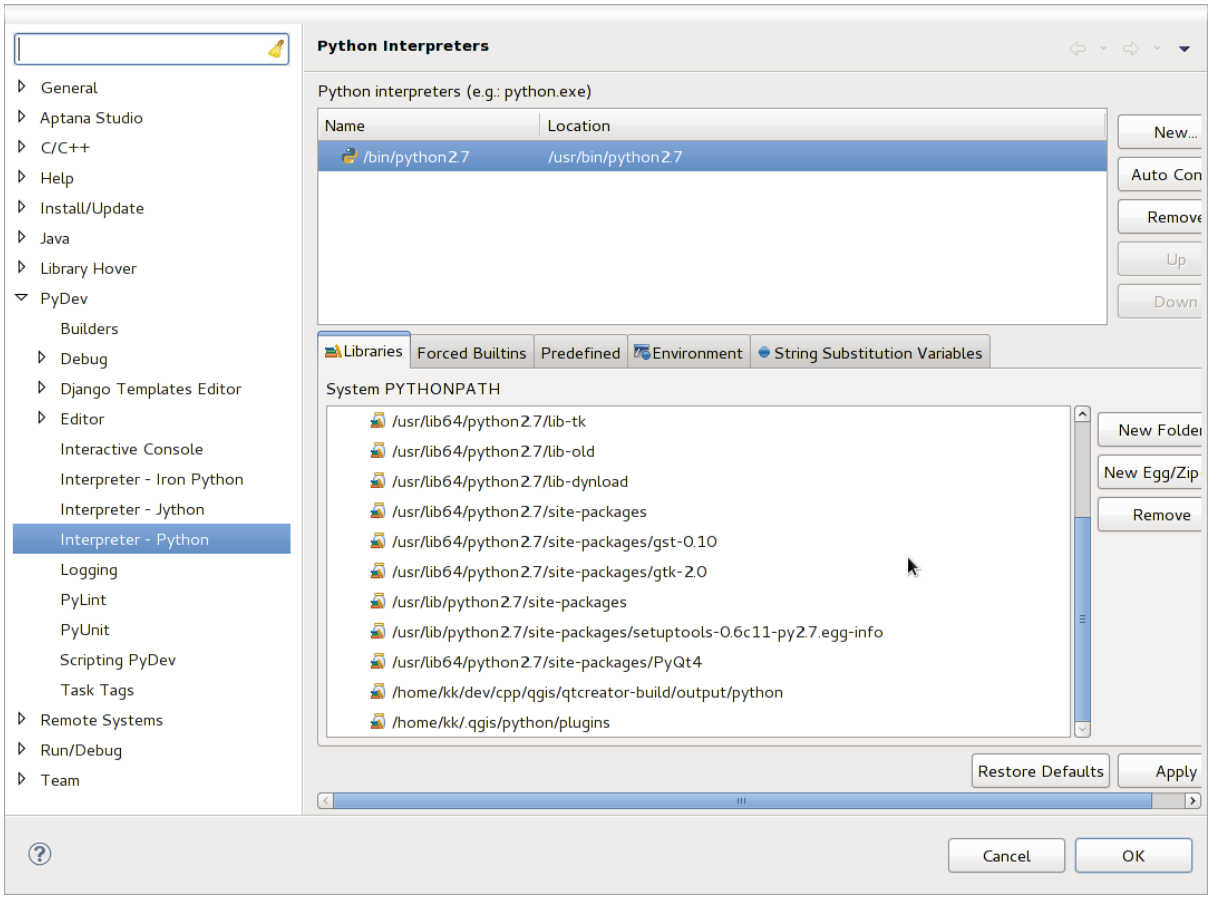


Fig. 16.4: Console di debug PyDev

2. Per prima cosa apri la scheda Library.
3. Aggiungi una nuova cartella e scegli la cartella python dell'installazione di QGIS. Se non sai dove si trova questa cartella (non è la cartella dei plugin):
   1. Apri QGIS
   2. Avviare la console python

3. Immetti `qgis`
4. e premi Invio. Ti verrà mostrato il modulo QGIS utilizzato e il suo percorso.
5. Togli la parte finale `/qgis/__init__.pyc` da questo percorso e otterrai il percorso desiderato.

4. Dovresti anche aggiungere qui la cartella dei plugin (si trova in `python/plugins` sotto la cartella user profile).
5. Passa quindi alla scheda *Forced Builtins*, fai clic su *New…* e inserisci `qgis`. In questo modo Eclipse analizzerà l'API di QGIS. Probabilmente vuoi che Eclipse conosca anche l'API di PyQt. Perciò aggiungi anche PyQt come builtin forzato. Probabilmente dovrebbe essere già presente nella scheda delle librerie.
6. Fai clic su *OK* e hai finito.

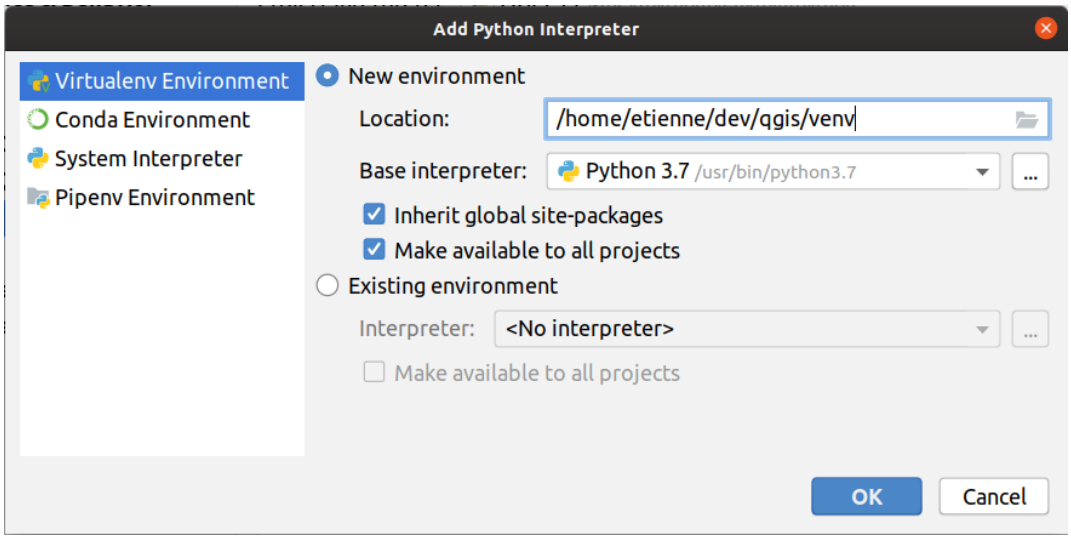
Nota: Ogni volta che l'API di QGIS cambia (ad esempio se stai compilando QGIS master e il file SIP è cambiato), dovresti tornare a questa pagina e fare semplicemente clic su *Apply*. In questo modo Eclipse analizzerà nuovamente tutte le librerie.

16.3.5 Debug con PyCharm su Ubuntu con un QGIS compilato

PyCharm è un IDE per Python sviluppato da JetBrains. Esiste una versione gratuita chiamata Community Edition e una a pagamento chiamata Professional. Puoi scaricare PyCharm dal sito web: https://www.jetbrains.com/pycharm/download.

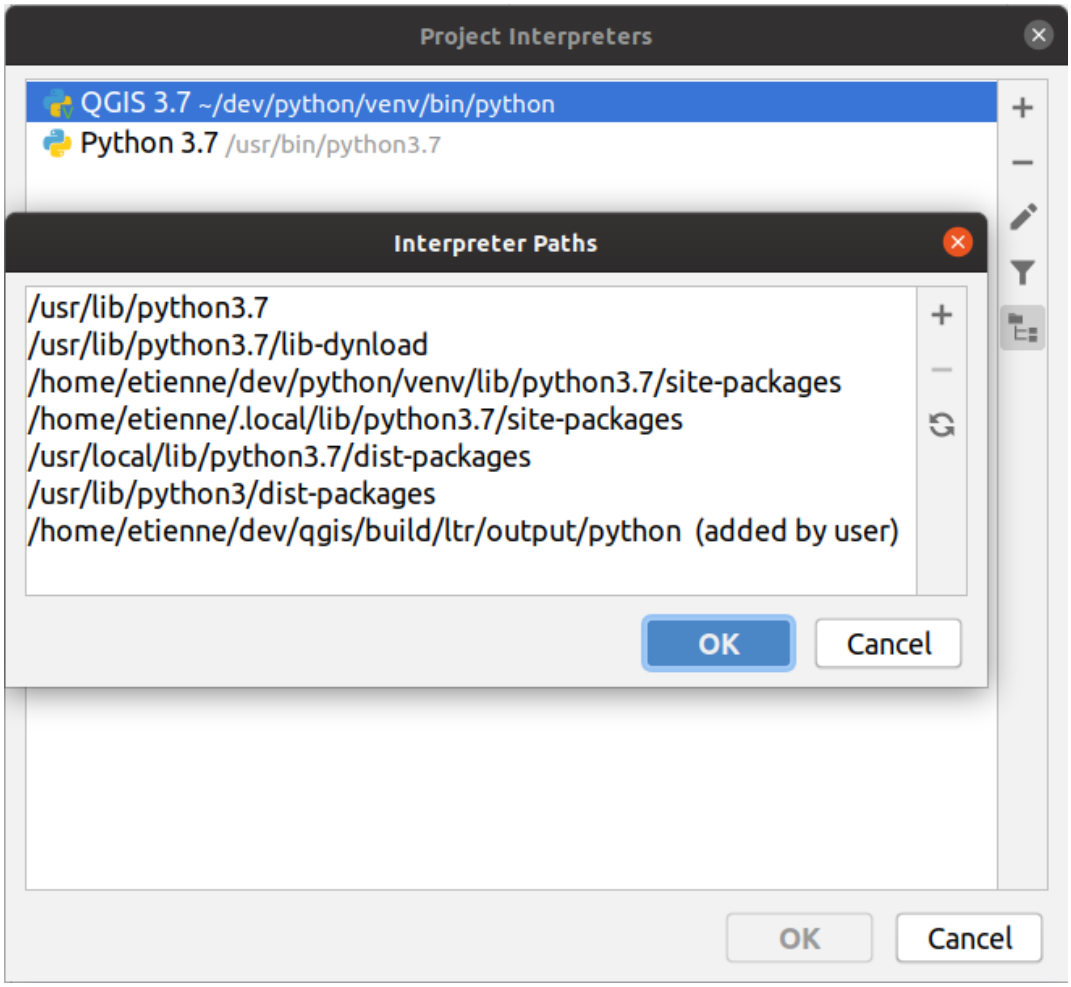
Si presume che sia stato compilato QGIS su Ubuntu con la directory di compilazione indicata `~/dev/qgis/build/master`. Non è obbligatorio avere un QGIS autocompilato, ma solo questo è stato testato. I percorsi devono essere adattati.

1. In PyCharm, nel tuo *Project Properties*, *Project Interpreter*, creeremo un ambiente virtuale Python chiamato `QGIS`.
2. Fai clic sul piccolo ingranaggio e quindi su *Aggiungi*.
3. Seleziona *Ambiente virtuale*.
4. Seleziona un percorso generico per tutti i progetti Python, come `~/dev/qgis/venv`, perché useremo questo interprete Python per tutti i nostri plugin.
5. Scegli un interprete di base Python 3 disponibile sul tuo sistema e seleziona le due opzioni successive *Eredita i pacchetti globali del sito* e *Metti a disposizione di tutti i progetti*.



1. Fai clic su *OK*, torna sull'ingranaggio piccolo e fai clic su *Mostra tutto*.

2. Nella nuova finestra, seleziona il nuovo interprete `QGIS` e fai clic sull'ultima icona del menu verticale *Mostra i percorsi per l'interprete selezionato.*
3. Infine, aggiungi il seguente percorso assoluto all'elenco `~/dev/qgis/build/master/output/python`.



1. Riavvia PyCharm e potrai iniziare a usare questo nuovo ambiente virtuale Python per tutti i tuoi plugin.

PyCharm conoscerà l'API di QGIS e anche l'API di PyQt se si usi Qt fornito da QGIS, come `from qgis.PyQt.QtCore import QDir`. Il completamento automatico dovrebbe funzionare e PyCharm può ispezionare il codice.

Nella versione professionale di PyCharm, il debug remoto funziona bene. Per l'edizione Community, il debug remoto non è disponibile. Puoi accedere solo a un debugger locale, il che significa che il codice deve essere eseguito *all'interno* di PyCharm (come script o unittest), non in QGIS stesso. Per il codice Python eseguito *in* QGIS, si può usare il plugin *First Aid* menzionato sopra.

16.3.6 Debug con PDB

Se non utilizzi un IDE come Eclipse o PyCharm, puoi eseguire il debug utilizzando PDB, seguendo questi passaggi.

1. Per prima cosa aggiungi questo codice nel punto in cui vuoi eseguire il debug

```
# 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. Esegui quindi QGIS dalla linea di comando.

   In Linux fai:

```
$ ./Qgis
```

   In macOS fai:

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

3. E quando l'applicazione raggiunge il punto di interruzione, puoi digitare nella console!

****DA FARE: ****
Aggiungere informazioni sui test

16.4 Rilasciare il tuo plugin

Una volta che il tuo plugin è pronto e pensi che possa essere utile a qualcuno, non esitare a caricarlo su *Repository ufficiale dei plugin Python*. In quella pagina puoi trovare anche le linee guida su come preparare il plugin per farlo funzionare bene con l'installatore di plugin. Oppure, nel caso in cui vuoi creare un tuo repository di plugin, crea un semplice file XML che elenchi i plugin e i relativi metadati.

Presta particolare attenzione ai seguenti suggerimenti:

16.4.1 Metadati e nomi

- evita di utilizzare un nome troppo simile a quello di plugin esistenti
- se il tuo plugin ha una funzionalità simile a quella di un plugin esistente, spiega le differenze nel campo About, in modo che l'utente sappia quale utilizzare senza doverlo installare e testare
- evita di ripetere «plugin» nel nome del plugin stesso
- utilizza il campo Descrizione nei metadati per una descrizione di 1 linea, il campo About per istruzioni più dettagliate
- includi un repository di codice, un bug tracker e una home page; questo aumenterà notevolmente le possibilità di collaborazione e può essere realizzato molto facilmente con una delle infrastrutture web disponibili (GitHub, GitLab, Bitbucket, ecc.).
- scegli con cura i tag: evita quelli poco informativi (ad es. vettoriali) e preferisci quelli già utilizzati da altri (vedi il sito web del plugin)
- aggiungi un'icona appropriata, non lasciare quella predefinita; vedi l'interfaccia di QGIS per un suggerimento sullo stile da usare

16.4.2 Codice e guida

- non includere file generati (ui_*.py, resources_rc.py, file di aiuto generati…) e roba inutile (ad esempio .gitignore) nel repository
- aggiungi il plugin al menu appropriato (Vector, Raster, Web, Database)
- quando è opportuno (plugin che eseguono analisi), valuta la possibilità di aggiungere il plugin come sottoplugin del framework Processing: ciò consentirà agli utenti di eseguirlo in batch, di integrarlo in flussi di lavoro più complessi e vi libererà dall'onere di progettare un'interfaccia
- includi almeno una documentazione minima e, se utile per il test e la comprensione, dei dati di esempio.

16.4.3 Repository ufficiale dei plugin Python

Puoi trovare il repository *ufficiale* dei plugin Python all'indirizzo https://plugins.qgis.org/.

Per utilizzare il repository ufficiale devi ottenere un OSGEO ID dal portale web OSGEO.

Una volta che hai caricato il tuo plugin, questo verrà approvato da un membro dello staff e ti verrà notificato.

****DA FARE: ****
Inserisci un link al documento di governance

Permessi

Queste regole sono state implementate nel repository ufficiale dei plugin:

- ogni utente registrato può aggiungere un nuovo plugin
- Gli utenti *staff* possono approvare o disapprovare tutte le versioni dei plugin.
- Gli utenti che hanno il permesso speciale *plugins.can_approve* fanno approvare automaticamente le versioni che caricano.
- Gli utenti che hanno il permesso speciale *plugins.can_approve* possono approvare le versioni caricate da altri, purché siano nell'elenco dei *proprietari* dei plugin.
- un determinato plugin può essere cancellato e modificato solo dagli utenti *staff* e dai *proprietari* di plugin
- se un utente senza il permesso *plugins.can_approve* carica una nuova versione, la versione del plugin viene automaticamente non approvata.

Gestione del Trust

I membri dello staff possono concedere *trust* ai creatori di plugin selezionati impostando il permesso *plugins.can_approve* attraverso l'applicazione front-end.

La visualizzazione dei dettagli del plugin offre collegamenti diretti per concedere la fiducia al creatore del plugin o ai *proprietari* del plugin.

Validazione

I metadati del plugin vengono importati e convalidati automaticamente dal pacchetto compresso quando il plugin viene caricato.

Ecco alcune regole di validazione che dovresti conoscere quando vuoi caricare un plugin sul repository ufficiale:

1. il nome della cartella principale contenente il plugin deve contenere solo caratteri ASCII (A-Z e a-z), cifre e i caratteri underscore (_) e meno (-), inoltre non può iniziare con una cifra
2. `metadata.txt` è richiesto
3. tutti i metadati richiesti elencati in *metadata table* devono essere presenti
4. il campo di metadati *version* deve essere univoco
5. a license file must be included, saved as `LICENSE` with no extension (i.e. not `LICENSE.txt` for example)

Struttura Plugin

Following the validation rules the compressed (.zip) package of your plugin must have a specific structure to validate as a functional plugin. As the plugin will be unzipped inside the users plugins folder it must have it's own directory inside the .zip file to not interfere with other plugins. Mandatory files are: `metadata.txt`, `__init__.py` and `LICENSE`. But it would be nice to have a `README` and of course an icon to represent the plugin. Following is an example of how a `plugin.zip` could look like.

```
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
```

È possibile creare plugin nel linguaggio di programmazione Python. In confronto ai classici plugin scritti in C++ questi dovrebbero essere più facili da scrivere, capire, mantenere e distribuire grazie alla natura dinamica del linguaggio Python.

I plugin Python sono elencati insieme ai plugin C++ nel gestore dei plugin di QGIS. Vengono cercati in `~/(UserProfile)/python/plugins` e in questi percorsi:

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

Per le definizioni di ~ e `(UserProfile)` vedi core_e_external_plugins.

Nota: Impostando *QGIS_PLUGINPATH* a un percorso di cartella esistente, puoi aggiungere questo percorso alla lista dei percorsi che vengono cercati per i plugin.

CAPITOLO 17

Scrivere un plugin di Processing

Dipendentemente dal tipo di plugin che stai per sviluppare, potrebbe essere una scelta conveniente di aggiungere le sue funzionalità quali algoritmo di Processing (o un set di essi). Ciò fornirebbe una migliore integrazione all'interno di QGIS, funzionalità aggiuntive (dal momento che può essere eseguito nei componenti di Processing, come il modellatore o l'interfaccia di elaborazione batch), e un tempo di sviluppo più spedito (dal momento che Processing si farà carico della gran parte del lavoro).

Per distribuire questi algoritmi, è necessario creare un nuovo plugin che li aggiunge al Processing Toolbox. Il plugin dovrebbe contenere un provider di algoritmi, che deve essere registrato quando il plugin viene istanziato.

17.1 Creazione da zero

Per creare un plugin da zero che contiene un provider di algoritmi, è possibile seguire questi passaggi utilizzando il Plugin Builder:

1. Installare il plugin **Plugin Builder**
2. Crea un nuovo plugin usando il Plugin Builder. Quando il Plugin Builder ti chiederà il modello da usare, seleziona «Sorgente di Processing».
3. Il plugin creato contiene una sorgente con un singolo algoritmo. Il file della sorgente e dell'algoritmo sono entrambi commentati e contengono informazioni su come modificare la sorgente e gli algoritmi aggiuntivi. Fai riferimento ad essi per maggiori informazioni.

17.2 Aggiornare un plugin

Se si desidera aggiungere il plugin esistente a Processing, è necessario aggiungere un po" di codice.

1. Nel tuo `metadata.txt` file, bisogna che aggiungi una variabile:

   ```
   hasProcessingProvider=yes
   ```

2. Nel file Python dove il tuo plugin è impostato tramite il metodo `initGui`, bisogna adattare qualche linea in questo modo:

```
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. Puoi creare una cartella `processing_provider` contenente tre files:
   - `__init__.py` con niente dentro. Ciò è necessario per produrre un package Python valido.
   - `provider.py` il quale creerà il Processing provider ed esporrà i tuoi algoritmi.

```
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` che contiene il file algoritmo di esempio. Copia/incolla il contenuto del file script template file e modificalo conformemente alle tue esigenze.

Dovresti avere un albero simile a questo:

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

1. Ora puoi ricaricare il tuo plugin in QGIS e dovresti vedere il tuo script di esempio nella lista di Processing toolbox e modeler.

CAPITOLO 18

Usare Plugin Layer

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

from qgis.PyQt.QtGui import QImage
```

Se il tuo plugin usa i suoi metodi per visualizzare un layer mappa, scrivere il proprio tipo di layer basato su `QgsPluginLayer` potrebbe essere il modo migliore per implementarlo.

18.1 Sottoclasse QgsPluginLayer

Di seguito è riportato un esempio di implementazione minima di QgsPluginLayer. È basato sul codice originale del plugin d'esempio Watermark.

Il visualizzatore personalizzato è il componente dell'implementazione che definisce la rappresentazione grafica sulla mappa.

```
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)
```

(continues on next page)

(continua dalla pagina precedente)

```
        painter.restore()
        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
```

Il layer plugin può essere aggiunto al progetto e al mappa come qualsiasi altro layer della mappa:

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

Quando si carica un progetto contenente un layer di questo tipo, è necessaria una classe factory:

```
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)
```

CAPITOLO 19

Libreria analisi di rete

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

La libreria di analisi di rete può essere utilizzata per:

- creare un grafico matematico a partire da dati geografici (layer vettoriali polilineari)
- implementare i metodi di base della teoria dei grafi (attualmente solo l'algoritmo di Dijkstra)

La libreria di analisi di rete è stata creata esportando le funzioni di base dal plugin principale di RoadGraph e ora puoi utilizzare i suoi metodi nei plugin o direttamente dalla console Python.

19.1 Informazioni generali

In breve, un caso d'uso tipico può essere descritto come segue:

1. creare un grafo da geodati (solitamente layer vettoriali polilineari)
2. eseguire l'analisi del grafo
3. utilizzare i risultati dell'analisi (ad esempio, visualizzarli)

19.2 Costruire un grafo

La prima cosa da fare è preparare i dati di input, cioè convertire un layer vettoriale in un grafo. Tutte le azioni successive utilizzeranno questo grafo, non il layer.

Come sorgente possiamo utilizzare qualsiasi layer vettoriale di polilinee. I nodi delle polilinee diventano vertici del grafo e i segmenti delle polilinee sono bordi del grafo. Se più nodi hanno le stesse coordinate, allora sono lo stesso vertice del grafo. Quindi due linee che hanno un nodo in comune diventano collegate tra loro.

Inoltre, durante la creazione del grafo è possibile «fissare» («legare») al layer del vettore di input un numero qualsiasi di punti aggiuntivi. Per ogni punto aggiuntivo verrà trovata una corrispondenza — con il vertice del grafo più vicino o con il bordo del grafo più vicino. In quest'ultimo caso, il bordo verrà diviso e verrà aggiunto un nuovo vertice.

Gli attributi del layer vettoriale e la lunghezza di un bordo possono essere utilizzati come proprietà di un bordo.

La conversione da un layer vettoriale a grafo viene effettuata utilizzando il modello di programmazione Builder. Un grafo viene costruito usando un cosiddetto Director. Per ora esiste un solo Director: `QgsVectorLayerDirector`. Il Director imposta le impostazioni di base che verranno utilizzate per costruire un grafico da un layer vettoriale di linee, utilizzato dal costruttore per creare il grafico. Attualmente, come nel caso del director, esiste un solo costruttore: `QgsGraphBuilder`, che crea oggetti `QgsGraph`. Puoi implementare i propri costruttori per creare un grafo compatibile con librerie come BGL o NetworkX.

Per calcolare le proprietà dei bordi si utilizza lo schema di programmazione strategy. Per ora sono disponibili solo le strategie `QgsNetworkDistanceStrategy` (che tiene conto della lunghezza del percorso) e `QgsNetworkSpeedStrategy` (che considera anche la velocità). Puoi implementare la tua strategia personale, che utilizzerà tutti i parametri necessari. Ad esempio, il plugin RoadGraph utilizza una strategia che calcola il tempo di percorrenza utilizzando la lunghezza dei bordi e il valore della velocità dagli attributi.

È il momento di addentrarsi nel processo.

Prima di tutto, per utilizzare questa libreria dobbiamo importare il modulo di analisi

```
from qgis.analysis import *
```

Poi alcuni esempi di creazione di un director

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)

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

Per costruire un director, occorre passare un layer vettoriale che sarà usato come fonte per la struttura del grafo e le informazioni sul movimento consentito su ogni segmento stradale (movimento unidirezionale o bidirezionale, direzione diretta o inversa). La call si presenta come segue

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

Ecco l'elenco completo del significato di questi parametri:

- `vectorLayer` — layer vettoriale usato per costruire il grafo

- `directionFieldId` — indice del campo della tabella degli attributi, dove sono memorizzate le informazioni sulla direzione delle strade. Se `-1`, non utilizzare affatto questa informazione. Un numero intero.
- `directDirectionValue` — Valore del campo per le strade con direzione diretta (che si muovono dal primo punto della linea all'ultimo). Una stringa.
- `reverseDirectionValue` — valore del campo per le strade con direzione inversa (che si muovono dall'ultimo punto della linea al primo). Una stringa.
- `bothDirectionValue` — valore del campo per le strade bidirezionali (per tali strade ci si può muovere dal primo punto all'ultimo e dall'ultimo al primo). Una stringa.
- `defaultDirection` — direzione predefinita della strada. Questo valore sarà utilizzato per le strade in cui il campo `directionFieldId` non è impostato o ha un valore diverso da uno dei tre valori specificati sopra. I valori possibili sono:
  - `QgsVectorLayerDirector.DirectionForward` — Diretto a senso unico
  - `QgsVectorLayerDirector.DirectionBackward` — Inversione unidirezionale
  - `QgsVectorLayerDirector.DirectionBoth` — bidirezionale

È quindi necessario creare una strategia per il calcolo delle proprietà dei bordi

```
# 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)
```

E informare il director di questa strategia

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

Ora possiamo usare il costruttore, che creerà il grafo. Il costruttore della classe `QgsGraphBuilder` utilizza diversi parametri:

- `crs` — sistema di riferimento delle coordinate da usare. Parametro obbligatorio.
- `otfEnabled` — utilizzare la riproiezione «al volo» o no. Per impostazione predefinita `True` (usa OTF).
- `topologyTolerance` — tolleranza topologica. Il valore predefinito è 0.
- `ellipsoid` — ellissoide da usare. Per impostazione predefinita «WGS84».

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

Inoltre, possiamo definire diversi punti che verranno utilizzati nell'analisi. Ad esempio

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

Ora tutto è a posto e possiamo costruire il grafo e «legare» questi punti ad esso

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

La costruzione del grafo può richiedere un certo tempo (che dipende dal numero di elementi in un layer e dalla dimensione del layer). `tiedPoints` è un elenco con le coordinate dei punti «legati». Quando l'operazione di costruzione è terminata, si può ottenere il grafico e utilizzarlo per l'analisi

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

Con il codice che segue possiamo ottenere gli indici dei vertici dei nostri punti

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

19.3 Analisi grafo

L'analisi di rete viene utilizzata per trovare le risposte a due domande: quali vertici sono connessi e come trovare il percorso più breve. Per risolvere questi problemi, la libreria di analisi di rete fornisce l'algoritmo di Dijkstra.

L'algoritmo di Dijkstra trova il percorso più breve da un vertice del grafo a tutti gli altri e i valori dei parametri di ottimizzazione. I risultati possono essere rappresentati come un albero del percorso più breve.

L'albero del percorso più breve è un grafo diretto pesato (o più precisamente un albero) con le seguenti proprietà:

- solo un vertice non ha bordi in entrata: la radice dell'albero.
- tutti gli altri vertici hanno un solo bordo in entrata
- se il vertice B è raggiungibile dal vertice A, allora il percorso da A a B è il solo percorso disponibile ed è ottimale (il più breve) su questo grafo

Per ottenere l'albero del percorso più breve, utilizzare i metodi `shortestTree()` e `dijkstra()` della classe `QgsGraphAnalyzer`. Si consiglia di utilizzare il metodo `dijkstra()` perché funziona più velocemente e utilizza la memoria in modo più efficiente.

Il metodo `shortestTree()` è utile quando si vuole percorrere l'albero del percorso più breve. Crea sempre un nuovo oggetto grafo (QgsGraph) e accetta tre variabili:

- `source` — grafo in ingresso
- `startVertexIdx` — indice del punto sull" albero (la radice dell'albero)
- `criterionNum` — numero di proprietà del bordo da usare (a partire da 0).

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

Il metodo `dijkstra()` ha gli stessi argomenti, ma restituisce due array. Il primo elemento *n* contiene l'indice del bordo entrante o -1 se non ci sono bordi entranti. Nel secondo elemento *n* contiene la distanza dalla radice dell'albero al vertice *n* o DOUBLE_MAX se il vertice *n* non è raggiungibile dalla radice.

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

Ecco un codice molto semplice per visualizzare l'albero del percorso più breve utilizzando il grafo creato con il metodo `shortestTree()` (seleziona il layer linestring nel pannello *Layer* e sostituisci le coordinate con le tue).

Avvertimento: Usa questo codice solo come esempio; crea molti oggetti `QgsRubberBand` e può essere lento su grandi insiemi di dati.

```
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())
```

(continues on next page)

(continua dalla pagina precedente)

```
12
13 pStart = QgsPointXY(1179661.925139,5419188.074362)
14 tiedPoint = director.makeGraph(builder, [pStart])
15 pStart = tiedPoint[0]
16
17 graph = builder.graph()
18
19 idStart = graph.findVertex(pStart)
20
21 tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)
22
23 i = 0
24 while (i < tree.edgeCount()):
25   rb = QgsRubberBand(iface.mapCanvas())
26   rb.setColor (Qt.red)
27   rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
28   rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
29   i = i + 1
```

Stessa cosa, ma utilizzando il metodo `dijkstra()`.

```
1  from qgis.core import *
2  from qgis.gui import *
3  from qgis.analysis import *
4  from qgis.PyQt.QtCore import *
5  from qgis.PyQt.QtGui import *
6
7  vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
   ↪'lines')
8
9  director = QgsVectorLayerDdirector(vectorLayer, -1, '', '', '',␣
   ↪QgsVectorLayerDirector.DirectionBoth)
10 strategy = QgsNetworkDistanceStrategy()
11 director.addStrategy(strategy)
12 builder = QgsGraphBuilder(vectorLayer.crs())
13
14 pStart = QgsPointXY(1179661.925139,5419188.074362)
15 tiedPoint = director.makeGraph(builder, [pStart])
16 pStart = tiedPoint[0]
17
18 graph = builder.graph()
19
20 idStart = graph.findVertex(pStart)
21
22 (tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)
23
24 for edgeId in tree:
25   if edgeId == -1:
26     continue
27   rb = QgsRubberBand(iface.mapCanvas())
28   rb.setColor (Qt.red)
29   rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
30   rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())
```

19.3.1 Trovare i percorsi più brevi

Per trovare il percorso ottimale tra due punti si utilizza il seguente approccio. Entrambi i punti (inizio A e fine B) sono «legati» al grafo quando viene costruito. Quindi, utilizzando il metodo `shortestTree()` o `dijkstra()` si costruisce l'albero del percorso più breve con radice nel punto iniziale A. Nello stesso albero si trova anche il punto finale B e si inizia a percorrere l'albero dal punto B al punto A. L'intero algoritmo può essere scritto come:

```
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
```

A questo punto abbiamo il percorso, sotto forma di elenco invertito di vertici (i vertici sono elencati in ordine inverso, dal punto finale al punto iniziale) che saranno visitati durante il percorso.

Ecco il codice di esempio per QGIS Python Console (potrebbe essere necessario caricare e selezionare un layer linestring in legenda e sostituire le coordinate nel codice con le tue) che utilizza il metodo `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)

(continua dalla pagina precedente)

```
    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)
```

Ed ecco lo stesso esempio, ma utilizzando il metodo `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)

(continua dalla pagina precedente)

```
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

19.3.2 Zone di disponibilità

La zona di disponibilità per il vertice A è il sottoinsieme dei vertici del grafo che sono accessibili dal vertice A e il costo dei percorsi da A a questi vertici non è superiore a un certo valore.

Questo può essere mostrato più chiaramente con il seguente esempio: «C'è una stazione dei vigili del fuoco. Quali zone della città può raggiungere un camion dei pompieri in 5 minuti? 10 minuti? 15 minuti?». Le risposte a queste domande sono le zone di disponibilità della caserma dei pompieri.

Per trovare le zone di disponibilità possiamo usare il metodo `dijkstra()` della classe `QgsGraphAnalyzer`. È sufficiente confrontare gli elementi dell'array cost con un valore predefinito. Se cost[i] è minore o uguale a un valore predefinito, allora il vertice i è all'interno della zona di disponibilità, altrimenti è fuori.

Un problema più difficile è quello di ottenere i confini della zona di disponibilità. Il confine inferiore è l'insieme dei vertici ancora accessibili, mentre il confine superiore è l'insieme dei vertici non accessibili. In realtà è semplice: si tratta del confine di disponibilità basato sui bordi dell'albero del cammino più breve per i quali il vertice di origine del bordo è accessibile e il vertice di destinazione del bordo no.

Ecco un esempio

```
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)

(continua dalla pagina precedente)

```
36
37 for i in upperBound:
38     centerPoint = graph.vertex(i).point()
39     rb = QgsRubberBand(iface.mapCanvas())
40     rb.setColor(Qt.red)
41     rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() - delta))
42     rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() - delta))
43     rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() + delta))
44     rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() + delta))
```

CAPITOLO 20

QGIS Server e Python

20.1 Introduzione

Per saperne di più su QGIS Server, leggi il QGIS-Server-manual.

QGIS Server è tre cose diverse:

1. Libreria QGIS Server: una libreria che fornisce un'API per la creazione di servizi web OGC.
2. QGIS Server FCGI: un'applicazione binaria FCGI `qgis_mapserv.fcgi` che, insieme a un server web, implementa una serie di servizi OGC (WMS, WFS, WCS ecc.) e API OGC (WFS3/OAPIF).
3. QGIS Development Server: un'applicazione binaria del server di sviluppo `qgis_mapserver` che implementa una serie di servizi OGC (WMS, WFS, WCS ecc.) e di API OGC (WFS3/OAPIF).

Questo capitolo del manuale si concentra sul primo argomento e, spiegando l'uso dell'API del server QGIS, mostra come sia possibile usare Python per estendere, migliorare o personalizzare il comportamento del server o come usare l'API del server QGIS per incorporare il server QGIS in un'altra applicazione.

Esistono diversi modi con cui puoi alterare il comportamento di QGIS Server o per estendere le sue capabilities per offrire nuovi servizi personalizzati o API:

- EMBEDDING → Usare l'API del server QGIS da un'altra applicazione Python
- STANDALONE → Eseguire QGIS Server come servizio WSGI/HTTP standalone
- FILTRI → Migliorare/Personalizzare QGIS Server con i plugin di filtro
- SERVICES → Aggiungere un nuovo *SERVICE*
- API OGC → Aggiungere una nuova *OGC API*

Le applicazioni embedding e standalone richiedono l'utilizzo dell'API Python di QGIS Server direttamente da un altro script o applicazione Python. Le altre opzioni sono più adatte quando vuoi aggiungere funzionalità personalizzate a un'applicazione binaria standard di QGIS Server (FCGI o server di sviluppo): in questo caso dovrai scrivere un plugin Python per l'applicazione server e registrare i tuoi filtri, servizi o API personalizzati.

20.2 Nozioni di base server API

Le classi fondamentali coinvolte in una tipica applicazione di QGIS Server sono:

- `QgsServer` l'istanza del server (in genere una singola istanza per tutta la durata dell'applicazione)
- `QgsServerRequest` l'oggetto richiesta (tipicamente ricreato ad ogni richiesta)
- `QgsServer.handleRequest(request, response)` elabora la richiesta e popola la risposta.

Il flusso di lavoro di QGIS Server FCGI o server di sviluppo può essere riassunto come segue:

```
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
```

All'interno del metodo `QgsServer.handleRequest(request, response)` vengono richiamate le callback dei plugin di filtro e `QgsServerRequest` e `QgsServerResponse` sono resi disponibili ai plugin attraverso la classe `QgsServerInterface`.

Avvertimento: Le classi del server QGIS non sono thread safe; per la creazione di applicazioni scalabili basate sull'API del server QGIS, devi utilizzare sempre un modello multiprocesso o dei container.

20.3 Standalone o embedding

Per le applicazioni server standalone o incorporate, è necessario utilizzare direttamente le classi server di cui sopra, impacchettandole in un'implementazione del server web che gestisce tutte le interazioni del protocollo HTTP con il client.

Segue un esempio minimo di utilizzo dell'API di QGIS Server (senza la parte HTTP):

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

(continues on next page)

(continua dalla pagina precedente)

```
23
24 app.exitQgis()
```

Ecco un esempio completo di applicazione standalone sviluppata per il test delle integrazioni continue sul repository del codice sorgente di QGIS, che mette in mostra un'ampia serie di diversi filtri per i plugin e schemi di autenticazione (non sono adatti alla produzione perché sono stati sviluppati solo a scopo di test, ma sono comunque interessanti per l'apprendimento): qgis_wrapped_server.py.

20.4 Plugin del server

I plugin python del server vengono caricati al momento dell'avvio dell'applicazione QGIS Server e possono essere utilizzati per registrare filtri, servizi o API.

La struttura di un plugin server è molto simile alla sua omologa desktop, un oggetto `QgsServerInterface` è reso disponibile ai plugin e questi ultimi possono registrare uno o più filtri, servizi o API personalizzati nel registro corrispondente, utilizzando uno dei metodi esposti dall'interfaccia server.

20.4.1 Plugin filtro server

I filtri sono disponibili in tre varianti e possono essere istanziati sottoclassando una delle classi seguenti e richiamando il metodo corrispondente di `QgsServerInterface`:

Tipo filtro	Classe base	Registrazione QgsServerInterface
I/O	`QgsServerFilter`	`registerFilter()`
Controllo accessi	`QgsAccessControlFilter`	`registerAccessControl()`
Cache	`QgsServerCacheFilter`	`registerServerCache()`

Filtri I/O

I filtri I/O possono modificare l'input e l'output del server (la richiesta e la risposta) dei servizi principali (WMS, WFS ecc.), consentendo di effettuare qualsiasi tipo di manipolazione del flusso di lavoro dei servizi. È possibile, ad esempio, limitare l'accesso a layer selezionati, inserire un foglio di stile XSL nella risposta XML, aggiungere un watermark a un'immagine WMS generata e così via.

A questo punto, ti può essere utile una rapida occhiata ai server plugins API docs.

Ogni filtro deve implementare almeno uno dei tre callback:

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

Tutti i filtri hanno accesso all'oggetto request/response (`QgsRequestHandler`) e possono manipolare tutte le sue proprietà (input/output) e sollevare eccezioni (anche se in modo piuttosto particolare, come vedremo più avanti).

Tutti questi metodi restituiscono un valore booleano che indica se la call deve essere propagata ai filtri successivi. Se uno di questi metodi restituisce `False`, la catena si ferma, altrimenti la call si propagherà al filtro successivo.

Ecco lo pseudo-codice che mostra come il server gestisce una richiesta tipica e quando vengono richiamati i callback del filtro:

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

(continues on next page)

(continua dalla pagina precedente)

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

I paragrafi seguenti descrivono in dettaglio i callback disponibili.

onRequestReady

Viene chiamato quando la richiesta è pronta: l'URL e i dati in entrata sono stati analizzati e prima di entrare nei servizi principali (WMS, WFS ecc.) si passa a questo punto in cui puoi manipolare l'input ed eseguire azioni come:

- autenticazione/autorizzazione
- redirects
- aggiungere/rimuovere alcuni parametri (ad esempio i nomi dei tipi)
- sollevare eccezioni

Potresti anche sostituire completamente un servizio principale cambiando il parametro **SERVICE** e quindi bypassando completamente il servizio principale (non che questo abbia molto senso, però).

onSendResponse

Viene richiamato ogni volta che un output parziale viene scaricato dal buffer di risposta (cioè a **FCGI** `stdout` se si usa il server fcgi) e da lì al client. Questo accade quando vengono trasmessi contenuti enormi (come WFS GetFeature). In questo caso `onSendResponse()` può essere chiamato più volte.

Da notare che se la risposta non è in streaming, `onSendResponse()` non sarà chiamato affatto.

In ogni caso, l'ultimo (o unico) blocco sarà inviato al client dopo una chiamata a `onResponseComplete()`.

Il ritorno di `False` impedisce il trasferimento dei dati al client. Questo è auspicabile quando un plugin vuole raccogliere tutti i blocchi di una risposta ed esaminare o modificare la risposta in `onResponseComplete()`.

onResponseComplete

Questo metodo viene richiamato una volta quando i servizi di base (se sono attivi) terminano il loro processo e la richiesta è pronta per essere inviata al client. Come discusso in precedenza, questo metodo sarà chiamato prima che l'ultimo (o unico) blocco di dati sia inviato al client. Per i servizi di streaming, potrebbero essere state chiamate più volte `onSendResponse()`.

`onResponseComplete()` è il luogo ideale per fornire l'implementazione di nuovi servizi (WPS o servizi personalizzati) e per eseguire la manipolazione diretta dell'output proveniente dai servizi principali (ad esempio per aggiungere una filigrana a un'immagine WMS).

Da notare che la restituzione di `False` impedirà ai plugin successivi di eseguire `onResponseComplete()` ma, in ogni caso, impedirà l'invio della risposta al client.

Generazione di eccezioni da parte di un plugin

C'è ancora del lavoro da fare su questo argomento: l'implementazione attuale può distinguere tra eccezioni gestite e non gestite impostando una proprietà `QgsRequestHandler` a un'istanza di QgsMapServiceException, in questo modo il codice principale C++ può catturare le eccezioni python gestite e ignorare quelle non gestite (o meglio: registrarle).

Questo approccio fondamentalmente funziona, ma non è molto «pythonic»: un approccio migliore sarebbe quello di sollevare eccezioni dal codice python e vederle confluire nel ciclo C++ per essere gestite lì.

Scrivere un server plugin

Un plugin server è un plugin Python standard di QGIS, come descritto in *Sviluppo di plugin Python*, che fornisce solo un'interfaccia aggiuntiva (o alternativa): un tipico plugin desktop di QGIS ha accesso all'applicazione QGIS attraverso l'istanza `QgisInterface`, un plugin server ha accesso solo a una `QgsServerInterface` quando viene eseguito nel contesto dell'applicazione QGIS Server.

Per far sì che QGIS Server sappia che un plugin ha un'interfaccia server, è necessaria una voce di metadati speciale (in `metadata.txt`):

```
server=True
```

Importante: Solo i plugin con i metadati `server=True` saranno caricati ed eseguiti da QGIS Server.

Il plugin di esempio qgis3-server-vagrant discusso qui (con molti altri) è disponibile su github, alcuni plugin per server sono anche pubblicati nel repository ufficiale dei QGIS plugins.

Plugin file

Ecco la struttura delle cartelle del nostro plugin server di esempio.

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

__init__.py

Questo file è richiesto dal sistema di importazione di Python. Inoltre, QGIS Server richiede che questo file contenga una funzione `serverClassFactory()`, che viene chiamata quando il plugin viene caricato in QGIS Server all'avvio del server. Riceve un riferimento all'istanza di `QgsServerInterface` e deve restituire l'istanza della classe del plugin. Ecco come appare il plugin di esempio `__init__.py`:

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

HelloServer.py

È qui che avviene la magia e questo è l'aspetto della magia: (ad esempio `HelloServer.py`)

Un plugin server consiste tipicamente in uno o più callback racchiusi in istanze di una `QgsServerFilter`.

Ogni `QgsServerFilter` implementa uno o più dei seguenti callback:

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

L'esempio seguente implementa un filtro minimo che stampa *HelloServer!* nel caso in cui il parametro **SERVICE** sia uguale a «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
```

I filtri devono essere registrati nella **serverIface** come nell'esempio seguente:

```
class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(serverIface), 100)
```

Il secondo parametro di `registerFilter()` imposta una priorità che definisce l'ordine delle callback con lo stesso nome (la priorità più bassa viene invocata per prima).

Utilizzando i tre callback, i plugin possono manipolare l'input e/o l'output del server in molti modi diversi. In ogni momento, l'istanza del plugin ha accesso alla `QgsRequestHandler` attraverso la `QgsServerInterface`. La classe `QgsRequestHandler` ha molti metodi che possono essere usati per modificare i parametri di ingresso prima di entrare nel nucleo di elaborazione del server (usando `requestReady()`) o dopo che la richiesta è stata elaborata dai servizi centrali (usando `sendResponse()`).

I seguenti esempi coprono alcuni casi d'uso comuni:

Modificare l'input

Il plugin di esempio contiene un esempio di test che modifica i parametri di input provenienti dalla stringa di query; in questo esempio un nuovo parametro viene inserito nella `parameterMap` (già analizzata), questo parametro è poi visibile dai servizi di base (WMS, ecc.); alla fine dell'elaborazione dei servizi di base controlliamo che il parametro sia ancora presente:

```
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
```

Questo è un estratto di ciò che vedi nel file di log:

```
 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
```

Nella riga evidenziata, la stringa «SUCCESS» indica che il plugin ha superato il test.

La stessa tecnica può essere sfruttata per utilizzare un servizio personalizzato al posto di quello principale: puoi ad esempio ignorare una richiesta **WFS SERVICE** o qualsiasi altra richiesta principale semplicemente cambiando il parametro **SERVICE** in qualcosa di diverso e il servizio principale verrà ignorato. Puoi quindi inserire i tuoi risultati personalizzati nell'output e inviarli al client (questo è spiegato più avanti).

Suggerimento: Se vuoi davvero implementare un servizio personalizzato, si raccomanda di sottoclassare `QgsService` e di registrare il tuo servizio su `registerFilter()` chiamando il suo `registerService(service)`.

Modifica o sostituzione del risultato

L'esempio del filtro watermark mostra come sostituire l'output WMS con una nuova immagine ottenuta aggiungendo un'immagine watermark all'immagine WMS generata dal servizio centrale WMS:

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

class WatermarkFilter(QgsServerFilter):

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

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (params.get('SERVICE').upper() == 'WMS' \
                and params.get('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.onResponseComplete: image ready %s" % request.parameter("FORMAT"))
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG" if "png" in request.parameter("FORMAT") else "JPG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
        return True
```

In questo esempio viene controllato il valore del parametro **SERVICE** e se la richiesta in arrivo è un **WMS GETMAP** e non sono state impostate eccezioni da un plugin precedentemente eseguito o dal servizio principale (WMS in questo caso), l'immagine generata da WMS viene recuperata dal buffer di output e viene aggiunta l'immagine del watermark. Il passo finale consiste nel cancellare il buffer di output e sostituirlo con la nuova immagine generata. Si noti che in una situazione reale si dovrebbe verificare anche il tipo di immagine richiesta, invece di supportare solo PNG o JPG.

Filtri controllo accesso

I filtri di controllo degli accessi danno allo sviluppatore un controllo a grana fine sui layer, sugli elementi e sugli attributi a cui si può accedere; le seguenti callback possono essere implementate in un filtro di controllo degli accessi:

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

- `cacheKey()`

Plugin file

Ecco la struttura dello cartella del nostro plugin di esempio:

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

__init__.py

Questo file è richiesto dal sistema di importazione di Python. Come per tutti i plugin del server QGIS, questo file contiene una funzione `serverClassFactory()`, che viene chiamata quando il plugin viene caricato in QGIS Server all'avvio. Riceve un riferimento a un'istanza di `QgsServerInterface` e deve restituire un'istanza della classe del plugin. Ecco come appare il plugin di esempio `__init__.py`:

```
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)

(continua dalla pagina precedente)

```
32      """ Register AccessControlFilter """
33      serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

Questo esempio offre un accesso completo a tutti.

Il ruolo del plugin è quello di sapere chi è connesso.

In tutti questi metodi abbiamo il parametro layer on per poter personalizzare la restrizione per layer.

layerFilterExpression

Si usa per aggiungere una Espressione per limitare i risultati.

Ad esempio, per limitare gli elementi in cui l'attributo ``role"" è uguale a ``user"".

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

layerFilterSubsetString

Come il precedente, ma utilizzando la `SubsetString` (eseguita nel database).

Ad esempio, per limitare gli elementi in cui l'attributo ``role"" è uguale a ``user"".

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

layerPermissions

Limitare l'accesso al layer.

Restituisce un oggetto di tipo `LayerPermissions()`, che ha le proprietà:

- `canRead` per vederlo in `GetCapabilities` e avere accesso in lettura.
- `canInsert` per poter inserire un nuovo elemento.
- `canUpdate` per poter aggiornare un elemento.
- `canDelete` per poter eliminare un elemento.

Ad esempio, per limitare l'accesso in sola lettura:

```
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

Si usa per limitare la visibilità di un sottoinsieme specifico di attributi.

Il parametro attributo restituisce l'insieme corrente degli attributi visibili.

Ad esempio, per nascondere l'attributo ``role"":

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

allowToEdit

Viene utilizzato per limitare la modifica di un sottoinsieme di elementi.

È utilizzato nel protocollo `WFS-Transaction`.

Ad esempio, per poter modificare solo gli elementi che hanno l'attributo `role` con il valore `user`:

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

cacheKey

QGIS Server mantiene una cache delle caratteristiche, quindi per avere una cache per role si può indicare role in questo metodo. Oppure restituire `None` per disabilitare completamente la cache.

20.4.2 Servizi personalizzati

In QGIS Server, i servizi principali come WMS, WFS e WCS sono implementati come sottoclassi di `QgsService`.

Per implementare un nuovo servizio che verrà eseguito quando il parametro della query string `SERVICE` corrisponde al nome del servizio, puoi implementare la tua `QgsService` e registrare il tuo servizio nel `serviceRegistry()` chiamando il suo `registerService(service)`.

Ecco un esempio di servizio personalizzato chiamato `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)

(continua dalla pagina precedente)

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

20.4.3 API personalizzate

In QGIS Server, le API OGC di base come OAPIF (alias WFS3) sono implementate come collezioni di `QgsServerOgcApiHandler` che sono registrate in un'istanza di `QgsServerOgcApi` (o della sua classe madre `QgsServerApi`).

Per implementare una nuova API che verrà eseguita quando il percorso dell'url corrisponde a un determinato URL, puoi implementare le tue istanze `QgsServerOgcApiHandler`, aggiungerle a una `QgsServerOgcApi` e registrare l'API nel `serviceRegistry()` chiamando il suo `registerApi(api)`.

Ecco un esempio di API personalizzata che verrà eseguita quando l'URL contiene `/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)

(continua dalla pagina precedente)

```
        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)
```

CAPITOLO 21

Istruzioni riassuntive di PyQGIS

Suggerimento: I frammenti di codice in questa pagina hanno bisogno delle seguenti importazioni se sei al di fuori della console di pyqgis:

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

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

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

21.1 Interfaccia utente

Cambiare aspetto e modalità di interazione

```
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())
```

Cambiare icona e titolo

```
from qgis.PyQt.QtGui import QIcon

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

(continues on next page)

(continua dalla pagina precedente)

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

21.2 Impostazioni

Ottenere l'elenco di QgsSettings

```
from qgis.core import QgsSettings

qs = QgsSettings()

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

21.3 Barre degli strumenti

Rimuovere la barra degli strumenti

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

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

Rimuovere la barra degli strumenti delle azioni

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

21.4 Menu

Rimuovere menu

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

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

21.5 Area della mappa

Accedere all'area mappa

```
canvas = iface.mapCanvas()
```

Cambiare il colore dell'area mappa

```
from qgis.PyQt.QtCore import Qt

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

Intervallo di aggiornamento della mappa

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

21.6 Layer

Aggiungere un layer vettoriale

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

Rendere layer attivo

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

Elencare tutti i layer

```
from qgis.core import QgsProject

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

Ottenere il nome dei layer

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

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

Altrimenti

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

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

Trovare il layer in base al nome

```
from qgis.core import QgsProject

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

```
layer name you like
```

Impostazione layer attivo

```
from qgis.core import QgsProject

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

Aggiornamento layer a intervalli

```
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)
```

Metodi di visualizzazione

```
dir(layer)
```

Aggiungere un nuovo elemento con il modulo per gli elementi

```
from qgis.core import QgsFeature, QgsGeometry

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

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

Aggiungere un nuovo elemento senza il modulo per gli elementi

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

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

Aggiungere elementi

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

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

Aggiungere elementi selezionati

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

Aggiungere id elementi selezionati

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

Creare un layer in memoria dagli id degli elementi selezionati

```
from qgis.core import QgsFeatureRequest

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

Selezionare geometria

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

```
10.000000, 10.000000
```

Spostare geometria

```
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)>
```

Imposta il SR

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

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

Consultare il SR

```
from qgis.core import QgsProject

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

Nascondere un campo colonna

```
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)

(continua dalla pagina precedente)

```
        else:
            continue
```

Layer da 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])
```

Caricare tutti i layer vettoriali da 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)
```

Caricare layer di tasselli (XYZ-Layer)

```
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')
```

Rimuovere tutti i layer

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

Rimuovere tutto

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

21.7 Indice dei contenuti

Accedere a layer selezionati

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

Rimuovere menu contestuale

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

21.8 Legenda avanzata

Nodo radice

```
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())
```

Accedere al primo nodo figlio

```
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
```

Cercare gruppi e nodi

```
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)

(continua dalla pagina precedente)

```

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
```

Cercare gruppo in base al nome

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

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

Cercare layer in base all'id

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

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

Aggiungere layer

```
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)
```

Aggiungere gruppo

```
from qgis.core import QgsLayerTreeGroup

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

Spostare layer caricato

```
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)
```

Spostare layer caricato in uno specifico gruppo

```
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)
```

Modificare visibilità del layer attivo

```
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)
```

Gruppo selezionato

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

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

```
False
```

Espandere nodo

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

Trucco nodo nascosto

```
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))
```

Informazioni sui nodi

```
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)
```

Rimuovere layer

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

Rimuovere gruppo

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

Crea un nuovo indice (legenda)

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

Spostare nodo

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

Rinominare nodo

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

21.9 Algoritmi di elaborazione

Elenco degli algoritmi

```
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
```

Ricevere aiuto per gli algoritmi

Selezione casuale

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

```
...
```

Eseguire l'algoritmo

In questo esempio, il risultato viene memorizzato in un layer temporaneo in memoria che viene aggiunto al progetto.

```
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'}
```

**Quanti algoritmi ci sono?

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

**Quanti fornitori ci sono?

```
from qgis.core import QgsApplication

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

**Quante espressioni ci sono?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

21.10 Decorazioni

CopyRight

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

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

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

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

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

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

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

```

(continues on next page)

(continua dalla pagina precedente)

```
    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 Compositore di stampe

Individuare il layout di stampa per nome

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

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

21.12 Sorgenti

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