PyQGIS 3.16 developer cookbook

QGIS Project

2021 년 11 월 17 일
Contents

1 소개
  1.1 Scripting in the Python Console .............................................. 2
  1.2 파이썬 플러그인 ...................................................................... 2
  1.3 Running Python code when QGIS starts .................................... 3
  1.4 파이썬 응용 프로그램 .......................................................... 3
  1.5 Technical notes on PyQt and SIP ............................................ 6

2 Loading Projects .......................................................................... 7
  2.1 Resolving bad paths ............................................................... 8

3 레이어 불러오기 ........................................................................... 9
  3.1 벡터 레이어 ............................................................................. 9
  3.2 레스터 레이어 ......................................................................... 12
  3.3 QgsProject instance ............................................................... 14

4 Accessing the Table Of Contents (TOC) ......................................... 17
  4.1 The QgsProject class ............................................................. 17
  4.2 QgsLayerTreeGroup class ....................................................... 18

5 레스터 레이어 사용 ...................................................................... 23
  5.1 레이어 상세 정보 ..................................................................... 23
  5.2 Renderers ................................................................................ 24
  5.3 값 조회 .................................................................................. 26

6 벡터 레이어 사용 ......................................................................... 29
  6.1 Retrieving information about attributes .................................... 30
  6.2 벡터 레이어 상의 반복 작업 ................................................... 31
  6.3 피처 선택 ............................................................................... 32
  6.4 벡터 레이어 수정 ..................................................................... 34
  6.5 공간 인덱스 사용 .................................................................... 38
  6.6 The QgsVectorLayerUtils class .............................................. 39
  6.7 Creating Vector Layers ........................................................... 39
  6.8 벡터 레이어의 표현 (심볼) ...................................................... 43
  6.9 남은 이야기들 ......................................................................... 52

7 도형 다루기 .................................................................................. 55
  7.1 도형 작성 ............................................................................... 55
<table>
<thead>
<tr>
<th>수록번호</th>
<th>제목</th>
<th>페이지</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>도형에 접근</td>
<td>56</td>
</tr>
<tr>
<td>7.3</td>
<td>도형 관계계산 및 연산</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>투영 지원</td>
<td>61</td>
</tr>
<tr>
<td>8.1</td>
<td>좌표계</td>
<td>61</td>
</tr>
<tr>
<td>8.2</td>
<td>CRS Transformation</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>Using the Map Canvas</td>
<td>65</td>
</tr>
<tr>
<td>9.1</td>
<td>레이어 캔버스 내장시키기</td>
<td>66</td>
</tr>
<tr>
<td>9.2</td>
<td>고무줄과 버텍스 마커</td>
<td>67</td>
</tr>
<tr>
<td>9.3</td>
<td>캔버스에서 레이어 도구 사용하기</td>
<td>68</td>
</tr>
<tr>
<td>9.4</td>
<td>사용자 지정 레이어 도구 작성</td>
<td>69</td>
</tr>
<tr>
<td>9.5</td>
<td>사용자 지정 레이어 캔버스 아이템 작성</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>렌더링 및 출력</td>
<td>73</td>
</tr>
<tr>
<td>10.1</td>
<td>단순 렌더링</td>
<td>73</td>
</tr>
<tr>
<td>10.2</td>
<td>서로 다른 좌표계를 가진 레이어들을 렌더링하기</td>
<td>74</td>
</tr>
<tr>
<td>10.3</td>
<td>Output using print layout</td>
<td>74</td>
</tr>
<tr>
<td>11</td>
<td>표현식을 이용한 속성값의 필터링 및 계산</td>
<td>79</td>
</tr>
<tr>
<td>11.1</td>
<td>표현식 파싱하기</td>
<td>80</td>
</tr>
<tr>
<td>11.2</td>
<td>표현식 평가하기</td>
<td>80</td>
</tr>
<tr>
<td>11.3</td>
<td>Handling expression errors</td>
<td>83</td>
</tr>
<tr>
<td>12</td>
<td>Reading And Storing Settings</td>
<td>85</td>
</tr>
<tr>
<td>13</td>
<td>사용자에게 정보 전달하기</td>
<td>89</td>
</tr>
<tr>
<td>13.1</td>
<td>Showing messages. The QgsMessageBar class</td>
<td>89</td>
</tr>
<tr>
<td>13.2</td>
<td>진행률 표시</td>
<td>91</td>
</tr>
<tr>
<td>13.3</td>
<td>로그 기록</td>
<td>93</td>
</tr>
<tr>
<td>14</td>
<td>Authentication infrastructure</td>
<td>97</td>
</tr>
<tr>
<td>14.1</td>
<td>소개</td>
<td>98</td>
</tr>
<tr>
<td>14.2</td>
<td>Glossary</td>
<td>98</td>
</tr>
<tr>
<td>14.3</td>
<td>QgsAuthManager the entry point</td>
<td>98</td>
</tr>
<tr>
<td>14.4</td>
<td>Adapt plugins to use Authentication infrastructure</td>
<td>102</td>
</tr>
<tr>
<td>14.5</td>
<td>Authentication GUIs</td>
<td>102</td>
</tr>
<tr>
<td>15</td>
<td>Tasks - doing heavy work in the background</td>
<td>107</td>
</tr>
<tr>
<td>15.1</td>
<td>소개</td>
<td>107</td>
</tr>
<tr>
<td>15.2</td>
<td>예제</td>
<td>108</td>
</tr>
<tr>
<td>16</td>
<td>파이썬 플러그인 개발</td>
<td>115</td>
</tr>
<tr>
<td>16.1</td>
<td>Structuring Python Plugins</td>
<td>115</td>
</tr>
<tr>
<td>16.2</td>
<td>코드 조각</td>
<td>124</td>
</tr>
<tr>
<td>16.3</td>
<td>플러그인 레이어 사용</td>
<td>127</td>
</tr>
<tr>
<td>16.4</td>
<td>IDE settings for writing and debugging plugins</td>
<td>128</td>
</tr>
<tr>
<td>16.5</td>
<td>Releasing your plugin</td>
<td>136</td>
</tr>
<tr>
<td>17</td>
<td>Writing a Processing plugin</td>
<td>141</td>
</tr>
<tr>
<td>17.1</td>
<td>Creating from scratch</td>
<td>141</td>
</tr>
<tr>
<td>17.2</td>
<td>Updating a plugin</td>
<td>142</td>
</tr>
<tr>
<td>18</td>
<td>네트워크 분석 라이브러리</td>
<td>145</td>
</tr>
<tr>
<td>18.1</td>
<td>일반 정보</td>
<td>145</td>
</tr>
<tr>
<td>18.2</td>
<td>그래프 만들기</td>
<td>146</td>
</tr>
</tbody>
</table>
18.3 그래프 분석 ................................................................. 148

19 QGIS Server and Python .................................................. 155
19.1 소개 ........................................................................ 155
19.2 Server API basics ..................................................... 156
19.3 Standalone or embedding ............................................. 156
19.4 Server plugins .......................................................... 157

20 Cheat sheet for PyQGIS .................................................... 169
20.1 User Interface ............................................................ 169
20.2 설정 .................................................................. 169
20.3 툴바 .................................................................. 170
20.4 Menus ................................................................ 170
20.5 Canvas ................................................................ 170
20.6 레이어 ................................................................ 171
20.7 Table of contents ........................................................ 174
20.8 Advanced TOC ........................................................ 175
20.9 Processing algorithms ............................................... 178
20.10 Decorators ............................................................ 179
20.11 Composer ............................................................... 180
20.12 Sources ................................................................. 181
This document is intended to be both a tutorial and a reference guide. While it does not list all possible use cases, it should give a good overview of the principal functionality.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section gnu_fdl.

This license also applies to all code snippets in this document.

Python support was first introduced in QGIS 0.9. There are several ways to use Python in QGIS Desktop (covered in the following sections):

- Issue commands in the Python console within QGIS
- Create and use plugins
- Automatically run Python code when QGIS starts
- Create processing algorithms
- Create functions for expressions in QGIS
- Create custom applications based on the QGIS API

Python bindings are also available for QGIS Server, including Python plugins (see QGIS Server and Python) and Python bindings that can be used to embed QGIS Server into a Python application.

There is a complete QGIS C++ API reference that documents the classes from the QGIS libraries. The Pythonic QGIS API (pyqgis) is nearly identical to the C++ API.

Another good resource for learning how to perform common tasks is to download existing plugins from the plugin repository and examine their code.
1.1 Scripting in the Python Console

QGIS provides an integrated Python console for scripting. It can be opened from the Plugins ▶️ Python Console menu:

The screenshot above illustrates how to get the layer currently selected in the layer list, show its ID and optionally, if it is a vector layer, show the feature count. For interaction with the QGIS environment, there is an iface variable, which is an instance of QgisInterface. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application.

For user convenience, the following statements are executed when the console is started (in the future it will be possible to set further initial commands)

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

For those which use the console often, it may be useful to set a shortcut for triggering the console (within Settings ▶️ Keyboard shortcuts...)

1.2 파이썬 플러그인

The functionality of QGIS can be extended using plugins. Plugins can be written in Python. The main advantage over C++ plugins is simplicity of distribution (no compiling for each platform) and easier development.

Many plugins covering various functionality have been written since the introduction of Python support. The plugin installer allows users to easily fetch, upgrade and remove Python plugins. See the Python Plugins page for more information about plugins and plugin development.

참고: Python plugins are also available for QGIS server. See QGIS Server and Python for further details.
1.3 Running Python code when QGIS starts

QGIS를 실행할 때마다 파이썬 코드를 실행시킬 수 있는 방법은 두 가지로 구분합니다.

1. Creating a startup.py script
2. Setting the PYQGIS_STARTUP environment variable to an existing Python file

1.3.1 startup.py 파일

Every time QGIS starts, the user’s Python home directory

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

is searched for a file named startup.py. If that file exists, it is executed by the embedded Python interpreter.

참고: The default path depends on the operating system. To find the path that will work for you, open the Python Console and run QStandardPaths.standardLocations(QStandardPaths.AppDataLocation) to see the list of default directories.

1.3.2 The PYQGIS_STARTUP environment variable

You can run Python code just before QGIS initialization completes by setting the PYQGIS_STARTUP environment variable to the path of an existing Python file.

This code will run before QGIS initialization is complete. This method is very useful for cleaning sys.path, which may have undesirable paths, or for isolating/loading the initial environment without requiring a virtual environment, e.g. homebrew or MacPorts installs on Mac.

1.4 파이썬 응용 프로그램

It is often handy to create scripts for automating processes. With PyQGIS, this is perfectly possible — import the qgis.core module, initialize it and you are ready for the processing.

Or you may want to create an interactive application that uses GIS functionality — perform measurements, export a map as PDF, … The qgis.gui module provides various GUI components, most notably the map canvas widget that can be incorporated into the application with support for zooming, panning and/or any further custom map tools.

PyQGIS custom applications or standalone scripts must be configured to locate the QGIS resources, such as projection information and providers for reading vector and raster layers. QGIS Resources are initialized by adding a few lines to the beginning of your application or script. The code to initialize QGIS for custom applications and standalone scripts is similar. Examples of each are provided below.

참고: Do not use qgis.py as a name for your script. Python will not be able to import the bindings as the script’s name will shadow them.
1.4.1 Using PyQGIS in standalone scripts

To start a standalone script, initialize the QGIS resources at the beginning of the script:

```python
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()
```

First we import the qgis.core module and configure the prefix path. The prefix path is the location where QGIS is installed on your system. It is configured in the script by calling the setPrefixPath() method. The second argument of setPrefixPath() is set to True, specifying that default paths are to be used.

The QGIS install path varies by platform; the easiest way to find it for your system is to use the Scripting in the Python Console from within QGIS and look at the output from running:

```python
QgsApplication.prefixPath()
```

After the prefix path is configured, we save a reference to QgsApplication in the variable qgs. The second argument is set to False, specifying that we do not plan to use the GUI since we are writing a standalone script. With QgsApplication configured, we load the QGIS data providers and layer registry by calling the initQgis() method.

```python
qgs.initQgis()
```

With QGIS initialized, we are ready to write the rest of the script. Finally, we wrap up by calling exitQgis() to remove the data providers and layer registry from memory.

```python
qgs.exitQgis()
```

1.4.2 Using PyQGIS in custom applications

The only difference between Using PyQGIS in standalone scripts and a custom PyQGIS application is the second argument when instantiating the QgsApplication. Pass True instead of False to indicate that we plan to use a GUI.

```python
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
```

(다음 페이지에 계속)
# 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()

Now you can work with the QGIS API - load layers and do some processing or fire up a GUI with a map canvas. The possibilities are endless :-)
• QGIS를 응용 프로그램과 함께 패키지하는 방법입니다. 응용 프로그램을 배포하는 데 더 많은 노력이 필요하고 용량이 더 커지게 되지만, 사용자가 추가로 다른 소프트웨어를 다운로드해 설치해야 하는 부담이 줄어들게 됩니다.

The two deployment models can be mixed. You can provide a standalone applications on Windows and macOS, but for Linux leave the installation of GIS up to the user and his package manager.

1.5 Technical notes on PyQt and SIP

We’ve decided for Python as it’s one of the most favoured languages for scripting. PyQGIS bindings in QGIS 3 depend on SIP and PyQt5. The reason for using SIP instead of the more widely used SWIG is that the QGIS code depends on Qt libraries. Python bindings for Qt (PyQt) are done using SIP and this allows seamless integration of PyQGIS with PyQt.

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import QgsProject,
               QgsPathResolver

from qgis.gui import QgsLayerTreeMapCanvasBridge,
```
Sometimes you need to load an existing project from a plugin or (more often) when developing a standalone QGIS Python application (see: 파이썬 응용 프로그램).

To load a project into the current QGIS application you need to create an instance of the QgsProject class. This is a singleton class, so you must use its instance() method to do it. You can call its read() method, passing the path of the project to be loaded:

```python
# 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())
```

ten data/01_project.qgs

If you need to make modifications to the project (for example to add or remove some layers) and save your changes, call the write() method of your project instance. The write() method also accepts an optional path for saving the project to a new location:

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

Both read() and write() functions return a boolean value that you can use to check if the operation was successful.

참고: If you are writing a QGIS standalone application, in order to synchronise the loaded project with the canvas you
need to instantiate a QgsLayerTreeMapCanvasBridge as in the example below:

```python
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 Resolving bad paths

It can happen that layers loaded in the project are moved to another location. When the project is loaded again all the layer paths are broken.

The QgsPathResolver class with the `setPathPreprocessor()` allows setting a custom path pre-processor function, which allows for manipulation of paths and data sources prior to resolving them to file references or layer sources.

The processor function must accept a single string argument (representing the original file path or data source) and return a processed version of this path.

The path pre-processor function is called **before** any bad layer handler.

Some use cases:

1. replace an outdated path:

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

2. replace a database host address with a new one:

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

3. replace stored database credentials with new ones:

   ```python
def my_processor(path):
      path = path.replace("user='gis_team'", "user='team_awesome'")
      path = path.replace("password='cats'", "password='g7as!m*'")
      return path
QgsPathResolver.setPathPreprocessor(my_processor)
```
The code snippets on this page need the following imports:

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

- 벡터 레이어
- 레스터 레이어
- *QgsProject instance*

데이터를 담고 있는 레이어를 열어봅시다. QGIS 는 벡터 및 레스터 레이어를 인식합니다. 추가적으로 사용자 정의 레이어도 이용할 수 있지만, 이 문서에서는 사용자 정의 레이어를 다루지 않습니다.

### 3.1 벡터 레이어

To create and add a vector layer instance to the project, specify the layer’s data source identifier, name for the layer and provider’s name:

```python
# get the path to the shapefile e.g. /home/project/data/ports.shp
path_to_airports_layer = "testdata/airports.shp"

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

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

(다음 페이지에 계속)
레이어 불러오기

For a geopackage vector layer:

```python
# get the path to a geopackage e.g. /usr/share/qgis/resources/data/world_map.gpkg
path_to_gpkg = os.path.join(QgsApplication.pkgDataPath(), "resources", "data", "world_map.gpkg")
# append the layername part
gpkg_countries_layer = path_to_gpkg + "|layername=countries"
# e.g. gpkg_places_layer = "/usr/share/qgis/resources/data/world_map.________gpkg|layername=countries"
vlayer = QgsVectorLayer(gpkg_countries_layer, "Countries 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`:

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

This creates a new layer and adds it to the current QGIS project (making it appear in the layer list) in one step. The function returns the layer instance or `None` if the layer couldn’t be loaded.

다음 목록은 벡터 데이터 제공자를 통해 여러 데이터 소스에 접근하는 방법을 보여줍니다.

- OGR library (Shapefile and many other file formats) — data source is the path to the file:
  - for Shapefile:

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

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

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

- PostGIS database - data source is a string with all information needed to create a connection to PostgreSQL database.

`QgsDataSourceUri` class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn’t available:

```python
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")

참고: The False argument passed to uri.uri(False) prevents the expansion of the authentication configu-
ration parameters, if you are not using any authentication configuration this argument does not make any difference.

• csv or other delimited text files — to open a file with a semicolon as a delimiter, with field 《x》 for X coordinate
and field 《y》 for Y coordinate you would use something like this:

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

참고: The provider string is structured as a URL, so the path must be prefixed with file://. Also it allows
WKT (well known text) formatted geometries as an alternative to x and y fields, and allows the coordinate reference
system to be specified. For example:

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

• GPX files — the 《gpx》 data provider reads tracks, routes and waypoints from gpx files. To open a file, the type
(track/route/waypoint) needs to be specified as part of the url:

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

• SpatiaLite database — Similarly to PostGIS databases, QgsDataSourceUri can be used for generation of data
source identifier:

uri = QgsDataSourceUri()
uri.setDatabase('/home/martin/test-2.3.sqlite')
schema = '
table = 'Towns'
geom_column = 'Geometry'
uri.setDataSource(schema, table, geom_column)
display_name = 'Towns'
vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
QgsProject.instance().addMapLayer(vlayer)

• MySQL WKB-based geometries, through OGR — data source is the connection string to the table:

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

• WFS connection: the connection is defined with a URI and using the WFS provider:

3.1. 벡터 레이어
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")

The uri can be created using the standard urllib library:

```python
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))
```

참고: You can change the data source of an existing layer by calling `setDataSource()` on a `QgsVectorLayer` instance, as in the following example:

```python
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 레스터 레이어

For accessing raster files, GDAL library is used. It supports a wide range of file formats. In case you have troubles with opening some files, check whether your GDAL has support for the particular format (not all formats are available by default). To load a raster from a file, specify its filename and display name:

```python
# 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!")
```

To load a raster from a geopackage:

```python
# 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!")
```
Similarly to vector layers, raster layers can be loaded using the addRasterLayer function of the QgisInterface object:

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

This creates a new layer and adds it to the current project (making it appear in the layer list) in one step.

To load a PostGIS raster:

PostGIS rasters, similar to PostGIS vectors, can be added to a project using a URI string. It is efficient to keep a re usable dictionary of strings for the database connection parameters. This makes it easy to edit the dictionary for the applicable connection. The dictionary is then encoded into a URI using the `postgresraster` provider metadata object. After that the raster can be added to the project.

```python
uri_config = {
    '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,
    'authcfg':QconfigId, # The QGIS authentication database ID holding connection details.
    'username':None, # The PostgreSQL user name.
    'password':None, # The PostgreSQL password for the user.
    'schema':'public', # The database schema that the table is located in.
    'table':'my_rasters', # The database table to be loaded.
    'geometrycolumn':'rast', # raster column in PostGIS table
    'sql':None, # An SQL WHERE clause. It should be placed at the end of the string.
    'key':None, # A key column from the table.
    'srid':None, # A string designating the SRID of the coordinate reference system.
    'estimatedmetadata':'False', # A boolean value telling if the metadata is estimated.
    'type':None, # A WKT string designating the WKB Type.
    'selectatid':None, # Set to True to disable selection by feature ID.
    'options':None, # other PostgreSQL connection options not in this list.
    'enableTime':None, # 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")
```

Raster layers can also be created from a WCS service:
Here is a description of the parameters that the WCS URI can contain:

WCS URI is composed of key=value pairs separated by &. It is the same format like query string in URL, encoded the same way. `QgsDataSourceUri` should be used to construct the URI to ensure that special characters are encoded properly.

- **url** (required): WCS Server URL. Do not use VERSION in URL, because each version of WCS is using different parameter name for `GetCapabilities` version, see param version.
- **identifier** (required): Coverage name
- **time** (optional): time position or time period (beginPosition/endPosition[/timeResolution])
- **format** (optional): Supported format name. Default is the first supported format with tif in name or the first supported format.
- **crs** (optional): CRS in form AUTHORITY:ID, e.g. EPSG:4326. Default is EPSG:4326 if supported or the first supported CRS.
- **username** (optional): Username for basic authentication.
- **password** (optional): Password for basic authentication.
- **IgnoreGetMapUrl** (optional, hack): If specified (set to 1), ignore GetCoverage URL advertised by GetCapabilities. May be necessary if a server is not configured properly.
- **InvertAxisOrientation** (optional, hack): If specified (set to 1), switch axis in GetCoverage request. May be necessary for geographic CRS if a server is using wrong axis order.
- **IgnoreAxisOrientation** (optional, hack): If specified (set to 1), do not invert axis orientation according to WCS standard for geographic CRS.
- **cache** (optional): cache load control, as described in QNetworkRequest::CacheLoadControl, but request is resend as PreferCache if failed with AlwaysCache. Allowed values: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. Default is AlwaysCache.

Alternatively you can load a raster layer from WMS server. However currently it’s not possible to access `GetCapabilities` response from API —you have to know what layers you want:

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

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

### 3.3 QgsProject instance

If you would like to use the opened layers for rendering, do not forget to add them to the `QgsProject` instance. The `QgsProject` instance takes ownership of layers and they can be later accessed from any part of the application by their unique ID. When the layer is removed from the project, it gets deleted, too. Layers can be removed by the user in the QGIS interface, or via Python using the `removeMapLayer()` method.

Adding a layer to the current project is done using the `addMapLayer()` method:
QgsProject.instance().addMapLayer(rlayer)

To add a layer at an absolute position:

```python
# 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))
```

If you want to delete the layer use the `removeMapLayer()` method:

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

In the above code, the layer id is passed (you can get it calling the `id()` method of the layer), but you can also pass the layer object itself.

For a list of loaded layers and layer ids, use the `mapLayers()` method:

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

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import (
    QgsProject,
    QgsVectorLayer,
)
```
레이어 불러오기
Accessing the Table Of Contents (TOC)

- *The QgsProject class*
- *QgsLayerTreeGroup class*

You can use different classes to access all the loaded layers in the TOC and use them to retrieve information:

- QgsProject
- QgsLayerTreeGroup

### 4.1 The QgsProject class

You can use QgsProject to retrieve information about the TOC and all the layers loaded.

You have to create an instance of QgsProject and use its methods to get the loaded layers.

The main method is `mapLayers()`. It will return a dictionary of the loaded layers:

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

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

The dictionary keys are the unique layer ids while the values are the related objects.

It is now straightforward to obtain any other information about the layers:

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

print(layers_list)

dict =
{'countries': <QgsVectorLayer: 'countries' (ogr)>}

You can also query the TOC using the name of the layer:

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

참고: A list with all the matching layers is returned, so we index with [0] to get the first layer with this name.

4.2 QgsLayerTreeGroup class

The layer tree is a classical tree structure built of nodes. There are currently two types of nodes: group nodes (QgsLayerTreeGroup) and layer nodes (QgsLayerTreeLayer).

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

The project layer tree can be accessed easily with the method layerTreeRoot() of the QgsProject class:

root = QgsProject.instance().layerTreeRoot()

root is a group node and has children:

root.children()

A list of direct children is returned. Sub group children should be accessed from their own direct parent.

We can retrieve one of the children:

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

<qgis._core.QgsLayerTreeLayer object at 0x7f1e1ea54168>

Layers can also be retrieved using their (unique) id:

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

And groups can also be searched using their names:

root.findGroup('Group Name')

QgsLayerTreeGroup has many other useful methods that can be used to obtain more information about the TOC:
# list of all the checked layers in the TOC
checked_layers = root.checkedLayers()
print(checked_layers)

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

Now let’s add some layers to the project’s layer tree. There are two ways of doing that:

1. **Explicit addition** using the `addLayer()` or `insertLayer()` functions:

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

2. **Implicit addition**: since the project’s layer tree is connected to the layer registry it is enough to add a layer to the map layer registry:

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

You can switch between `QgsVectorLayer` and `QgsLayerTreeLayer` easily:

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

Layer node: <qgis._core.QgsLayerTreeLayer object at 0x7f24423175e0>
Map layer: <QgsVectorLayer: 'countries' (ogr)>

Groups can be added with the `addGroup()` method. In the example below, the former will add a group to the end of the TOC while for the latter you can add another group within an existing one:

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

To moving nodes and groups there are many useful methods.

Moving an existing node is done in three steps:

1. cloning the existing node
2. moving the cloned node to the desired position
3. deleting the original node

```python
# 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)
```

It is a little bit more *complicated* to move a layer around in the legend:
```python
# 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)

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

or moving it to an existing group:

```python
# 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)
```

Some other methods that can be used to modify the groups and layers:

```python
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)
```

The code snippets on this page need the following imports if you’re outside the pyqgis console:

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

(다음 페이지에 계속)
from qgis.PyQt.QtGui import { 
    QColor,
}
CHAPTER 5

래스터 레이어 사용

5.1 레이어 상세 정보

A raster layer consists of one or more raster bands —referred to as single band and multi band rasters. One band represents a matrix of values. A color image (e.g. aerial photo) is a raster consisting of red, blue and green bands. Single band rasters typically represent either continuous variables (e.g. elevation) or discrete variables (e.g. land use). In some cases, a raster layer comes with a palette and the raster values refer to the colors stored in the palette.

The following code assumes rlayer is a QgsRasterLayer object.

```python
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 : 20.839452843000001, -33.7507750070000014

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

23
# get the total band count of the raster
print(rlayer.bandCount())
1

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

Band 1: Height

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

<qgis._core.QgsLayerMetadata object at 0x13711d558>

## 5.2 Renderer

When a raster layer is loaded, it gets a default renderer based on its type. It can be altered either in the layer properties or programmatically.

To query the current renderer:

```
print(rlayer.renderer())

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

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

To set a renderer, use the `setRenderer()` method of `QgsRasterLayer`. There are a number of renderer classes (derived from `QgsRasterRenderer`):

- `QgsHillshadeRenderer`
- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`
- `QgsRasterContourRenderer`
- `QgsSingleBandColorDataRenderer`
- `QgsSingleBandGrayRenderer`
- `QgsSingleBandPseudoColorRenderer`

Single band raster layers can be drawn either in gray colors (low values = black, high values = white) or with a pseudocolor algorithm that assigns colors to the values. Single band rasters with a palette can also be drawn using the palette. Multiband layers are typically drawn by mapping the bands to RGB colors. Another possibility is to use just one band for drawing.
5.2.1 단일 밴드 래스터

Let’s say we want a render single band raster layer with colors ranging from green to yellow (corresponding to pixel values from 0 to 255). In the first stage we will prepare a QgsRasterShader object and configure its shader function:

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

The shader maps the colors as specified by its color map. The color map is provided as a list of pixel values with associated colors. There are three modes of interpolation:

- **linear (Interpolated):** the color is linearly interpolated from the color map entries above and below the pixel value
- **discrete (Discrete):** the color is taken from the closest color map entry with equal or higher value
- **exact (Exact):** the color is not interpolated, only pixels with values equal to color map entries will be drawn

In the second step we will associate this shader with the raster layer:

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

The number 1 in the code above is the band number (raster bands are indexed from one).

Finally we have to use the `triggerRepaint()` method to see the results:

```python
rlayer.triggerRepaint()
```

5.2.2 다중 밴드 래스터

By default, QGIS maps the first three bands to red, green and blue to create a color image (this is the MultiBandColor drawing style. In some cases you might want to override these setting. The following code interchanges red band (1) and green band (2): 

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

In case only one band is necessary for visualization of the raster, single band drawing can be chosen, either gray levels or pseudocolor.

We have to use `triggerRepaint()` to update the map and see the result:

```python
rlayer_multi.triggerRepaint()
```
5.3 값 조회

Raster values can be queried using the `sample()` method of the `QgsRasterDataProvider` class. You have to specify a `QgsPointXY` and the band number of the raster layer you want to query. The method returns a tuple with the value and `True` or `False` depending on the results:

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

Another method to query raster values is using the `identify()` method that returns a `QgsRasterIdentifyResult` object.

```python
ident = rlayer.dataProvider().identify(QgsPointXY(20.5, -34), QgsRaster.IdentifyFormatValue)
if ident.isValid():
    print(ident.results())
```

```python
{1: 323.0}
```

In this case, the `results()` method returns a dictionary, with band indices as keys, and band values as values. For instance, something like `{1: 323.0}`

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
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)
```

(다음 페이지에 계속)
(이전 페이지에서 계속)

```python
QColor,
```

5.3. 값 조회
CHAPTER 6

벡터 레이어 사용

• Retrieving information about attributes
  • 벡터 레이어 상의 반복 작업
  • 피처 선택
    - 속성에 접근하기
    - 선택된 피처에 대한 반복 작업
    - 피처 부분 집합에 대한 반복 작업
  • 벡터 레이어 수정
    - 피처 추가
    - 피처 삭제
    - 피처 수정
    - 편집 버퍼로 벡터 레이어 수정
    - 항목 추가 및 제거
  • 공간 인덱스 사용
• The QgsVectorLayerUtils class
• Creating Vector Layers
  - From an instance of QgsVectorFileWriter
  - Directly from features
  - From an instance of QgsVectorLayer
• 벡터 레이어의 표현 (심볼)
  - 단일 심볼 렌더러
6.1 Retrieving information about attributes

You can retrieve information about the fields associated with a vector layer by calling `fields()` on a `QgsVectorLayer` object:

```python
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
for field in vlayer.fields():
    print(field.name(), field.typeName())
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Integer64</th>
</tr>
</thead>
<tbody>
<tr>
<td>fk_region</td>
<td>Integer64</td>
</tr>
<tr>
<td>ELEV</td>
<td>Real</td>
</tr>
<tr>
<td>NAME</td>
<td>String</td>
</tr>
<tr>
<td>USE</td>
<td>String</td>
</tr>
</tbody>
</table>

The `displayField()` and `mapTipTemplate()` methods of the `QgsVectorLayer` class provide information on the field and template used in the maptips tab.

When you load a vector layer, a field is always chosen by QGIS as the Display Name, while the HTML Map Tip is empty by default. With these methods you can easily get both:

```python
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
print(vlayer.displayField())
```

NAME

참고: If you change the Display Name from a field to an expression, you have to use `displayExpression()` instead of `displayField()`.
Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. The `layer` variable is assumed to have a `QgsVectorLayer` object.

```python
# "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())
    # fetch 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()
    print(attrs)
    # attrs is a list. It contains all the attribute values of this feature

break
```

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

In QGIS desktop, features can be selected in different ways: the user can click on a feature, draw a rectangle on the map canvas or use an expression filter. Selected features are normally highlighted in a different color (default is yellow) to draw user’s attention on the selection.

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

To select all the features, the `selectAll()` method can be used:

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

To select using an expression, use the `selectByExpression()` method:

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

To change the selection color you can use `setSelectionColor()` method of `QgsMapCanvas` as shown in the following example:

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

To add features to the selected features list for a given layer, you can call `select()` passing to it the list of features IDs:

```python
selected_fid = []
# Get the first feature id from the layer
for feature in layer.getFeatures():
    selected_fid.append(feature.id())
    break
# Add these features to the selected list
layer.select(selected_fid)
```

To clear the selection:

```python
layer.removeSelection()
```

6.3.1 속성에 접근하기

Attributes can be referred to by their name:

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

First feature

Alternatively, attributes can be referred to by index. This is a bit faster than using the name. For example, to get the second attribute:

```python
print(feature[1])
```
If you only need selected features, you can use the `selectedFeatures()` method from the vector layer:

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

If you want to iterate over a given subset of features in a layer, such as those within a given area, you have to add a `QgsFeatureRequest` object to the `getFeatures()` call. Here’s an example:

```python
areaOfInterest = QgsRectangle(450290, 400520, 450750, 400780)
request = QgsFeatureRequest().setFilterRect(areaOfInterest)
for feature in layer.getFeatures(request):
    # do whatever you need with the feature
    pass
```

For the sake of speed, the intersection is often done only using feature’s bounding box. There is however a flag `ExactIntersect` that makes sure that only intersecting features will be returned:

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

With `setLimit()` you can limit the number of requested features. Here’s an example:

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

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the examples above, you can build a `QgsExpression` object and pass it to the `QgsFeatureRequest` constructor. Here’s an example:

```python
# 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)
```

See 表현식을 이용한 속성값의 필터링 및 계산 for the details about the syntax supported by `QgsExpression`.

---

**6.3. 기초 작업**

6.3.2 선택된 피처에 대한 반복 작업

If you only need selected features, you can use the `selectedFeatures()` method from the vector layer:

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

6.3.3 피쳐 부분 집합에 대한 반복 작업

If you want to iterate over a given subset of features in a layer, such as those within a given area, you have to add a `QgsFeatureRequest` object to the `getFeatures()` call. Here’s an example:

```python
areaOfInterest = QgsRectangle(450290, 400520, 450750, 400780)
request = QgsFeatureRequest().setFilterRect(areaOfInterest)
for feature in layer.getFeatures(request):
    # do whatever you need with the feature
    pass
```

For the sake of speed, the intersection is often done only using feature’s bounding box. There is however a flag `ExactIntersect` that makes sure that only intersecting features will be returned:

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

With `setLimit()` you can limit the number of requested features. Here’s an example:

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

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the examples above, you can build a `QgsExpression` object and pass it to the `QgsFeatureRequest` constructor. Here’s an example:

```python
# 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)
```

See 表현식을 이용한 속성값의 필터링 및 계산 for the details about the syntax supported by `QgsExpression`.

---

6.3. 피처 선택 33
# 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 벡터 레이어 수정

Most vector data providers support editing of layer data. Sometimes they support just a subset of possible editing actions. Use the capabilities() function to find out what set of functionality is supported.

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

The layer supports DeleteFeatures

For a list of all available capabilities, please refer to the API Documentation of QgsVectorDataProvider.

To print layer’s capabilities textual description in a comma separated list you can use capabilitiesString() as in the following example:

```python
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'
```

参考: If you are working inside QGIS (either from the console or from a plugin), it might be necessary to force a redraw of the map canvas in order to see the changes you’ve done to the geometry, to the style or to the attributes:

```python
# 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 피처 추가

Create some QgsFeature instances and pass a list of them to provider’s addFeatures() method. It will return two values: result (True or False) and list of added features (their ID is set by the data store).

To set up the attributes of the feature, you can either initialize the feature passing a QgsFields object (you can obtain that from the fields() method of the vector layer) or call initAttributes() passing the number of fields you want to be added.

```python
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 피처 삭제

To delete some features, just provide a list of their feature IDs.

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

6.4.3 피처 수정

It is possible to either change feature’s geometry or to change some attributes. The following example first changes values of attributes with index 0 and 1, then it changes the feature’s geometry.

```python
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 })
```

팁: Favor QgsVectorLayerEditUtils class for geometry-only edits

If you only need to change geometries, you might consider using the QgsVectorLayerEditUtils which provides some useful methods to edit geometries (translate, insert or move vertex, etc.).
When editing vectors within QGIS application, you have to first start editing mode for a particular layer, then do some modifications and finally commit (or rollback) the changes. All the changes you make are not written until you commit them—they stay in layer’s in-memory editing buffer. It is possible to use this functionality also programmatically—it is just another method for vector layer editing that complements the direct usage of data providers. Use this option when providing some GUI tools for vector layer editing, since this will allow user to decide whether to commit/rollback and allows the usage of undo/redo. When changes are committed, all changes from the editing buffer are saved to data provider.

The methods are similar to the ones we have seen in the provider, but they are called on the `QgsVectorLayer` object instead.

For these methods to work, the layer must be in editing mode. To start the editing mode, use the `startEditing()` method. To stop editing, use the `commitChanges()` or `rollback()` methods. The first one will commit all your changes to the data source, while the second one will discard them and will not modify the data source at all.

To find out whether a layer is in editing mode, use the `isEditable()` method.

Here you have some examples that demonstrate how to use these editing methods.

```python
from qgis.PyQt.QtCore import QVariant

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

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

# delete a feature with specified ID
layer.deleteFeature(fid)

# set new geometry (QgsGeometry instance) for a feature
geometry = QgsGeometry.fromWkt("POINT(7 45)")
layer.changeGeometry(fid, geometry)

# update an attribute with given field index (int) to a given value
fieldIndex = 1
value = 'My new name'
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QVariant.String))

# remove a field
layer.deleteAttribute(fieldIndex)
```

In order to make undo/redo work properly, the above mentioned calls have to be wrapped into undo commands. (If you do not care about undo/redo and want to have the changes stored immediately, then you will have easier work by editing with data provider.)

Here is how you can use the undo functionality:

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

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

if problem_occurred:
    layer.destroyEditCommand()
    layer.deleteFeature(fid)

# ... tell the user that there was a problem
```

(다음 페이지에 계속)
The `beginEditCommand()` method will create an internal `Active` command and will record subsequent changes in vector layer. With the call to `endEditCommand()` the command is pushed onto the undo stack and the user will be able to undo/redo it from GUI. In case something went wrong while doing the changes, the `destroyEditCommand()` method will remove the command and rollback all changes done while this command was active.

You can also use the `with edit(layer)`-statement to wrap commit and rollback into a more semantic code block as shown in the example below:

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

This will automatically call `commitChanges()` in the end. If any exception occurs, it will `rollback()` all the changes. In case a problem is encountered within `commitChanges()` (when the method returns False) a `QgsEditError` exception will be raised.

### 6.4.5 항목 추가 및 제거

필드 (속성)를 추가하려면, 필드를 정의한 리스트를 지정해야 합니다. 필드를 삭제하려면 필드 인덱스 목록만 넘겨주면 됩니다.

```python
from qgis.PyQt.QtCore import QVariant

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

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

데이터 제공자에서 필드를 추가 또는 제거한 다음 레이어의 필드를 업데이트해야 합니다. 이는 변경 사항이 자동적으로 반영되지 않기 때문입니다.

### 6.4. 벡터 레이어 수정
layer.updateFields()

*Tip: Directly save changes using `with` based command*

Using `with edit(layer):` the changes will be committed automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollBack()` all the changes. See 편집 버퍼로 벡터 레이어 수정.

### 6.5 공간 인덱스 사용

벡터 레이어에 대해 자주 퀄리티 해야 할 경우 공간 인덱스를 사용하면 코드 실행 속도를 획기적으로 향상시킬 수 있습니다. 여러분이 보건 알코올을 작성하는데, 보건값을 계산하기 위해 주어진 위치에서 가장 가까운 포인트 레이어의 포인트 10개를 알아내야한다고 해보면 상상해보시죠. 공간 인덱스가 없다면, QGIS가 그 포인트 10개를 찾는 유일한 방법은 해당 위치에서 모든 포인트까지의 거리를 각각 계산한 다음 그 거리를 비교하는 것입니다. 특히 이 작업을 몇 군데의 위치에 대해 반복해야 할 경우 시간이 아주 오래 걸릴 수 있습니다. 레이어가 공간 인덱스를 가지고 있다면, 훨씬 효율적으로 작업할 수 있습니다.

공간 인덱스가 없는 레이어는 전화번호가 정렬되지도 색인되지도 않은 전화번호부라고 생각해보십시오. 어떤 사람의 전화번호를 찾으려면 처음부터 그 번호를 찾아가야 할 수밖에 없습니다.

Spatial indexes are not created by default for a QGIS vector layer, but you can create them easily. This is what you have to do:

- create spatial index using the `QgsSpatialIndex` class:

```python
index = QgsSpatialIndex()
```

- add features to index — index takes `QgsFeature` object and adds it to the internal data structure. You can create the object manually or use one from a previous call to the provider’s `getFeatures()` method.

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

- alternatively, you can load all features of a layer at once using bulk loading

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

- 공간 인덱스가 채워지면 쿼리를 해보십시오.

```python
# 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))
```

You can also use the `QgsSpatialIndexKDBush` spatial index. This index is similar to the standard `QgsSpatialIndex` but:

- supports only single point features
- is `static` (no additional features can be added to the index after the construction)
- is much faster!
- allows direct retrieval of the original feature’s points, without requiring additional feature requests
- supports true `distance based` searches, i.e. return all points within a radius from a search point
6.6 The QgsVectorLayerUtils class

The QgsVectorLayerUtils class contains some very useful methods that you can use with vector layers.

For example the `createFeature()` method prepares a QgsFeature to be added to a vector layer keeping all the eventual constraints and default values of each field:

```python
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
feat = QgsVectorLayerUtils.createFeature(vlayer)
```

The `getValues()` method allows you to quickly get the values of a field or expression:

```python
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
# select only the first feature to make the output shorter
vlayer.selectByIds([1])
val = QgsVectorLayerUtils.getValues(vlayer, "NAME", selectedOnly=True)
print(val)
```

6.7 Creating Vector Layers

There are several ways to generate a vector layer dataset:

- the QgsVectorFileWriter class: A convenient class for writing vector files to disk, using either a static call to writeAsVectorFormat() which saves the whole vector layer or creating an instance of the class and issue calls to addFeature(). This class supports all the vector formats that OGR supports (GeoPackage, Shapefile, GeoJSON, KML and others).

- the QgsVectorLayer class: instantiates a data provider that interprets the supplied path (url) of the data source to connect to and access the data. It can be used to create temporary, memory-based layers (memory) and connect to OGR datasets (ogr), databases (postgres, spatialite, mysql, mssql) and more (wfs, gpx, delimitedtext...).

6.7.1 From an instance of QgsVectorFileWriter

```python
# SaveVectorOptions contains many settings for the writer process
save_options = QgsVectorFileWriter.SaveVectorOptions()
transform_context = QgsProject.instance().transformContext()
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
                             "testdata/my_new_file.gpkg",
                             transform_context,
                             save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
else:
    print(error)
```

```python
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "ESRI Shapefile"
```
You can also convert fields to make them compatible with different formats by using the `FieldValueConverter`. For example, to convert array variable types (e.g. in Postgres) to a text type, you can do the following:

```python
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, 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, self, field):
        idx = self.layer.fields().indexFromName(field.name())
        if idx == self.list_field_idx:
            return QgsField(LIST_FIELD_NAME, QVariant.String)
        else:
            return None
```

(다음 페이지에 계속)
A destination CRS may also be specified—if a valid instance of `QgsCoordinateReferenceSystem` is passed as the fourth parameter, the layer is transformed to that CRS.

For valid driver names please call the `supportedFiltersAndFormats()` method or consult the supported formats by OGR—you should pass the value in the《Code》column as the driver name.

Optionally you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes…There are a number of other (optional) parameters; see the `QgsVectorFileWriter` documentation for details.

### 6.7.2 Directly from features

```python
from qgis.PyQt.QtCore import QVariant

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

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

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

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

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

# add a feature
```

6.7. Creating Vector Layers
PyQGIS 3.16 developer cookbook

6.7.3 From an instance of QgsVectorLayer

Among all the data providers supported by the QgsVectorLayer class, let's focus on the memory-based layers. Memory provider is intended to be used mainly by plugin or 3rd party app developers. It does not store data on disk, allowing developers to use it as a fast backend for some temporary layers.

이 제공자는 문자열 (string), 정수 (int), 더블형 실수 (double) 유형의 필드를 지원합니다.

The memory provider also supports spatial indexing, which is enabled by calling the provider's createSpatialIndex() function. Once the spatial index is created you will be able to iterate over features within smaller regions faster (since it's not necessary to traverse all the features, only those in specified rectangle).

A memory provider is created by passing "memory" as the provider string to the QgsVectorLayer constructor.

The constructor also takes a URI defining the geometry type of the layer, one of: "Point", "LineString", "Polygon", "MultiPoint", "MultiLineString", "MultiPolygon" or "None".

이 URI 로 좌표계, 항목, 메모리기반 제공자의 인덱스 작업도 설정할 수 있습니다. 문법은 다음과 같습니다.

- crs=definition Specifies the coordinate reference system, where definition may be any of the forms accepted by QgsCoordinateReferenceSystem.createFromString()
- index=yes 제공자가 공간 인덱스를 사용하도록 설정합니다.
- field=name:type(length,precision) 레이어의 속성을 설정합니다. 속성의 명칭은 필수적이며, 선택적으로 유형 (정수, 더블형 실수, 문자 스트링), 길이 및 정밀도를 설정할 수 있습니다. 여러 개의 속성 항목을 정의할 수도 있습니다.

다음은 이 모든 옵션들을 포함하는 URI 의 예시입니다.

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

다음은 메모리기반 제공자를 생성하고 값을 채우는 코드의 예시입니다.

```python
from qgis.PyQt.QtCore import QVariant

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

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

# add a feature
fet = QgsFeature()
```
fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])
# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()

마지막으로, 모든 작업이 성공적이었는지 확인해봅시다.

# 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 ['Johny', 2, 0.3] <QgsPointXY: POINT(10 10)>

6.8 벡터 레이어의 표현 (심볼)

벡터 레이어를 렌더링할 때, 레이어와 관련된 렌더러와 심볼이 레이어의 표현을 결정합니다. 심볼은 피처의 시각적 표현을 그리는 일을 담당하는 클래스이며, 렌더러는 특정 피처에 대해 어떤 심볼을 적용할지 결정합니다.

The renderer for a given layer can be obtained as shown below:

renderer = layer.renderer()

그리고 이를 참고해 조금 더 나가봅시다.

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

Type: singleSymbol

There are several known renderer types available in the QGIS core library:

<table>
<thead>
<tr>
<th>유형</th>
<th>클래스</th>
<th>설명</th>
</tr>
</thead>
<tbody>
<tr>
<td>singleSymbol</td>
<td>QgsSingleSymbolRenderer</td>
<td>모든 피처를 동일한 심볼로 렌더링합니다.</td>
</tr>
<tr>
<td>categorizedSymbol</td>
<td>QgsCategorizedSymbolRenderer</td>
<td>피처를 각 카테고리에 대해 서로 다른 심볼로 렌더링합니다.</td>
</tr>
<tr>
<td>graduatedSymbol</td>
<td>QgsGraduatedSymbolRenderer</td>
<td>피처의 값의 범위에 따라 서로 다른 심볼로 렌더링합니다.</td>
</tr>
</tbody>
</table>

There might be also some custom renderer types, so never make an assumption there are just these types. You can query the application’s QgsRendererRegistry to find out currently available renderers:

6.8. 벡터 레이어의 표현 (심볼)
You can get the symbol used for rendering by calling `symbol()` method and change it with `setSymbol()` method (note for C++ devs: the renderer takes ownership of the symbol.)

You can change the symbol used by a particular vector layer by calling `setSymbol()` passing an instance of the appropriate symbol instance. Symbols for point, line and polygon layers can be created by calling the `createSimple()` function of the corresponding classes `QgsMarkerSymbol`, `QgsLineSymbol` and `QgsFillSymbol`.

The dictionary passed to `createSimple()` sets the style properties of the symbol.

For example you can replace the symbol used by a particular point layer by calling `setSymbol()` passing an instance of a `QgsMarkerSymbol`, as in the following code example:

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

name 은 마커의 형태를 의미하며, 다음 중 어느 것이라도 가능합니다.

- circle
- square
- cross
- rectangle
- diamond
- pentagon
- triangle
- equilateral_triangle
- star
- regular_star
- arrow
- filled_arrowhead
- x

To get the full list of properties for the first symbol layer of a symbol instance you can follow the example code:
print(layer.renderer().symbol().symbolLayers()[0].properties())

{"angle": '0', 'color': '255,0,0,255', 'horizontal_anchor_point': '1', 'joinstyle':
    'bevel', 'name': 'square', 'offset': '0,0', 'offset_map_unit_scale': '3x:0,0,0,0,0,
    'offset_unit': 'MM', 'outline_color': '35,35,35,255', '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'}

This can be useful if you want to alter some properties:

```python
# 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 카테고리 심볼 렌더러

When using a categorized renderer, you can query and set the attribute that is used for classification: use the `classAttribute()` and `setClassAttribute()` methods.

카테고리 목록을 얻는 방법은 다음과 같습니다.

```python
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()))
```

Where `value()` is the value used for discrimination between categories, `label()` is a text used for category description and `symbol()` method returns the assigned symbol.

The renderer usually stores also original symbol and color ramp which were used for the classification: `sourceColorRamp()` and `sourceSymbol()` methods.
이 렌더러는 앞에서 설명한 카테고리 심볼 렌더러와 매우 비슷하지만, 범주 당 하나의 속성을 할당하는 대신 값의 범위에 따라 할당하기 때문에 숫자 속성에만 사용할 수 있습니다.

이 렌더러가 사용하는 범위에 대해 상세히 조회하려면 다음과 같은 방법을 사용합니다.

```python
1 graduated_renderer = QgsGraduatedSymbolRenderer()

# Add a few categories
2 graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class 0-100', 0, 100), QgsMarkerSymbol()))
3 graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class 101-200', 101, 200), QgsMarkerSymbol()))

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

사용자 정의 등급 심볼 렌더러를 생성하고자 할 경우 다음 예시처럼 하면 됩니다. (이 예시에서는 범주 2 개인 간단한 배열을 생성합니다.)

```python
from qgis.PyQt import QtGui

myVectorLayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
myTargetField = 'ELEV'
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)

# Make our second symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eff')
mySymbol2 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
mySymbol2 setColor(myColour)
mySymbol2 setOpacity(myOpacity)
```

You can again use the `classAttribute()` (to find the classification attribute name), `sourceSymbol()` and `sourceColorRamp()` methods. Additionally there is the `mode()` method which determines how the ranges were created: using equal intervals, quantiles or some other method.

(다음 페이지에 계속)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('myRangeList', myRangeList)
myClassificationMethod = QgsApplication.classificationMethodRegistry().method("EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)
myVectorLayer.setRenderer(myRenderer)

6.8.4 심볼 다루기

For representation of symbols, there is QgsSymbol base class with three derived classes:

- QgsMarkerSymbol — for point features
- QgsLineSymbol — for line features
- QgsFillSymbol — for polygon features

Every symbol consists of one or more symbol layers (classes derived from QgsSymbolLayer). The symbol layers do the actual rendering, the symbol class itself serves only as a container for the symbol layers.

Having an instance of a symbol (e.g. from a renderer), it is possible to explore it: the type() method says whether it is a marker, line or fill symbol. There is a dump() method which returns a brief description of the symbol. To get a list of symbol layers:

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

0: SimpleMarker

To find out symbol’s color use color() method and setColor() to change its color. With marker symbols additionally you can query for the symbol size and rotation with the size() and angle() methods. For line symbols the width() method returns the line width.

기본적으로 크기 및 두께는 밀리미터 단위이며, 각도는 도 단위입니다.

심볼 레이어 다루기

As said before, symbol layers (subclasses of QgsSymbolLayer) determine the appearance of the features. There are several basic symbol layer classes for general use. It is possible to implement new symbol layer types and thus arbitrarily customize how features will be rendered. The layerType() method uniquely identifies the symbol layer class — the basic and default ones are SimpleMarker, SimpleLine and SimpleFill symbol layers types.

You can get a complete list of the types of symbol layers you can create for a given symbol layer class with the following code:

```python
from qgis.core import QgsSymbolLayerRegistry
myRegistry = QgsApplication.symbolLayerRegistry()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbol.Marker):
    print(item)
```
The `QgsSymbolLayerRegistry` class manages a database of all available symbol layer types.

To access symbol layer data, use its `properties()` method that returns a key-value dictionary of properties which determine the appearance. Each symbol layer type has a specific set of properties that it uses. Additionally, there are the generic methods `color()`, `size()`, `angle()` and `width()`, with their setter counterparts. Of course size and angle are available only for marker symbol layers and width for line symbol layers.

사용자 지정 심볼 레이어 생성

데이터를 어떻게 렌더링할지 사용자가 지정하고 싶다고 상상해보시오. 사용자가 원하는 방식대로 피처를 그려는, 자신만의 심볼 레이어 클래스를 생성할 수 있습니다. 다음은 지정된 반지름으로 빨간색 원을 그리는 마커의 예시 코드입니다.

```python
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)
```

The `layerType()` method determines the name of the symbol layer; it has to be unique among all symbol layers. The `properties()` method is used for persistence of attributes. The `clone()` method must return a copy of the symbol.
layer with all attributes being exactly the same. Finally there are rendering methods: \texttt{startRender()} is called before rendering the first feature, \texttt{stopRender()} when the rendering is done, and \texttt{renderPoint()} is called to do the rendering. The coordinates of the point(s) are already transformed to the output coordinates.

For polylines and polygons the only difference would be in the rendering method: you would use \texttt{renderPolyline()} which receives a list of lines, while \texttt{renderPolygon()} receives a list of points on the outer ring as the first parameter and a list of inner rings (or None) as a second parameter.

 일반적으로 사용자가 표현을 변경할 수 있도록 심볼 레이어 유형의 속성을 설정하기 위한 GUI 를 추가하는 것이 편리합니다. 앞의 예시에서 사용자가 원의 반경을 설정하도록 할 수 있습니다. 다음은 그런 위젯을 구현하는 예시 코드입니다.

```python
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()"))
```

This widget can be embedded into the symbol properties dialog. When the symbol layer type is selected in symbol properties dialog, it creates an instance of the symbol layer and an instance of the symbol layer widget. Then it calls the \texttt{setSymbolLayer()} method to assign the symbol layer to the widget. In that method the widget should update the UI to reflect the attributes of the symbol layer. The \texttt{symbolLayer()} method is used to retrieve the symbol layer again by the properties dialog to use it for the symbol.

On every change of attributes, the widget should emit the \texttt{changed()} signal to let the properties dialog update the symbol preview.

이제 마지막 단계만 남았습니다. QGIS 가 이 세 클래스들을 인식하도록 만드는 일입니다. 심볼 레이어를 레이어스트리에 추가하면 됩니다. 레이어스트리에 추가하지 않고 심볼 레이어를 사용할 수도 있지만, 예를 들어 사용자가 지정 심볼 레이어를 담고 있는 프로젝트 파일을 불러온다면 QGIS 에서 레이어 속성을 편집할 수 없는 등 몇몇 기능들을 사용할 수 없게 됩니다. 심볼 레이어에 대한 메타데이터도 생성해야 합니다.
from qgis.core import QgsSymbol, QgsSymbolLayerAbstractMetadata,
    QgsSymbolLayerRegistry

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):
    def __init__(self):
        super().__init__("FooMarker", "My new Foo marker", QgsSymbol.Marker)
    def createSymbolLayer(self, props):
        radius = float(props["radius"]) if "radius" in props else 4.0
        return FooSymbolLayer(radius)

fslmetadata = FooSymbolLayerMetadata()

QgsApplication.symbolLayerRegistry().addSymbolLayerType(fslmetadata)

You should pass layer type (the same as returned by the layer) and symbol type (marker/line/fill) to the constructor of the parent class. The createSymbolLayer() method takes care of creating an instance of symbol layer with attributes specified in the props dictionary. And there is the createSymbolLayerWidget() method which returns the settings widget for this symbol layer type.

마지막 단계는 이 심볼 레이어를 레지스트리에 추가하는 일입니다. 이제 모두 끝났습니다.

### 6.8.5 사용자 정의 렌더러 생성

피처를 렌더링하는 데 어떻게 심볼을 선택할지에 대한 규칙을 마음대로 지정하고 싶을 경우 새로운 렌더러를 만드는 것이 유용할 수도 있습니다. 여러 항목을 조합해서 심볼을 결정해야 하거나, 현재 축척에 따라 심볼 크기를 변형해야 하는 등의 경우에 새로운 렌더러를 만들면 좋습니다.

다음 예시 코드는 마커 심볼 2개를 생성해서 각 피처마다 임의로 2개 중 1개를 선택하는 간단한 사용자 지정 렌더러입니다.

```python
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:
```

(다음 페이지에 계속)
The constructor of the parent QgsFeatureRenderer class needs a renderer name (which has to be unique among renderers). The symbolForFeature() method is the one that decides what symbol will be used for a particular feature. startRender() and stopRender() take care of initialization/finalization of symbol rendering. The usedAttributes() method can return a list of field names that the renderer expects to be present. Finally, the clone() method should return a copy of the renderer.

Like with symbol layers, it is possible to attach a GUI for configuration of the renderer. It has to be derived from QgsRendererWidget. The following sample code creates a button that allows the user to set the first symbol

```python
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
```

The constructor receives instances of the active layer (QgsVectorLayer), the global style (QgsStyle) and the current renderer. If there is no renderer or the renderer has different type, it will be replaced with our new renderer, otherwise we will use the current renderer (which has already the type we need). The widget contents should be updated to show current state of the renderer. When the renderer dialog is accepted, the widget’s renderer() method is called to get the current renderer—it will be assigned to the layer.

마지막으로 렌더러의 메타데이터를 생성하고 레지스트리에 렌더러를 등록해야 합니다. 이렇게 하지 않으면 렌더러와 함께 레이어를 불러올 수 없고, 사용자가 렌더러 목록에서 렌더러를 선택할 수 없습니다. 이제 RandomRenderer 에서 코드를 완성해봅시다.

```python
from qgis.core import (QgsRendererAbstractMetadata,
                      QgsRendererRegistry,
```

(다음 페이지에 계속)
PyQGIS 3.16 developer cookbook

```python
QgsApplication

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

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

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

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

Similarly as with symbol layers, abstract metadata constructor awaits renderer name, name visible for users and optionally name of renderer’s icon. The `createRenderer()` method passes a QDomElement instance that can be used to restore the renderer’s state from the DOM tree. The `createRendererWidget()` method creates the configuration widget. It does not have to be present or can return None if the renderer does not come with GUI.

To associate an icon with the renderer you can assign it in the `QgsRendererAbstractMetadata` constructor as a third (optional) argument —the base class constructor in the `RandomRendererMetadata` __init__() function becomes

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

The icon can also be associated at any later time using the `setIcon()` method of the metadata class. The icon can be loaded from a file (as shown above) or can be loaded from a Qt resource (PyQt5 includes .qrc compiler for Python).

### 6.9 남은 이야기들

**TODO:**

- creating/modifying symbols
- working with style (`QgsStyle`)
- working with color ramps (`QgsColorRamp`)
- exploring symbol layer and renderer registries

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import
    QgsGeometry,
    QgsPoint,
    QgsPointXY,
    QgsWkbTypes,
    QgsProject,
    QgsFeatureRequest,
```

(다음 페이지에 계속)
QgsVectorLayer,
QgsDistanceArea,
QgsUnitTypes,
QgsCoordinateTransform,
QgsCoordinateReferenceSystem
]
도형 다루기

• 도형 작성
• 도형에 접근
• 도형 관계계산 및 연산

Points, linestrings and polygons that represent a spatial feature are commonly referred to as geometries. In QGIS they are represented with the `QgsGeometry` class.

도형 한 개가 실제로는 단순(단일 영역, single-part) 도형의 집합인 경우가 종종 있습니다. 이런 도형을 다중 영역(multi-part) 도형이라고 합니다. 다중 영역 도형이 한 가지 유형의 단순 도형으로만 이루어져 있을 경우 다중 포인트, 다중 라인스트링, 다중 폴리곤이라 부릅니다. 예를 들어 여러 개의 섬으로 이루어진 국가라면 다중 폴리곤으로 표현할 수 있습니다.

도형의 좌표는 어떤 좌표계(CRS)라도 될 수 있습니다. 레이어에서 피처를 불러올 때, 해당 도형은 레이어의 좌표계를 따르는 좌표를 가지게 됩니다.

Description and specifications of all possible geometries construction and relationships are available in the OGC Simple Feature Access Standards for advanced details.

7.1 도형 작성

PyQGIS provides several options for creating a geometry:

- 좌표로부터
  ```python
  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),
  (다음 페이지에 계속)]]
  ```
Coordinates are given using QgsPoint class or QgsPointXY class. The difference between these classes is that QgsPoint supports M and Z dimensions.

A Polyline (Linestring) is represented by a list of points.

A Polygon is represented by a list of linear rings (i.e. closed linestrings). The first ring is the outer ring (boundary), optional subsequent rings are holes in the polygon. Note that unlike some programs, QGIS will close the ring for you so there is no need to duplicate the first point as the last.

다중 영역 도형은 한 단계 심화됩니다. 다중 포인트는 포인트의 목록, 다중 라인스트링은 라인스트링의 목록, 다중 폴리곤은 폴리곤의 목록입니다.

- WKT(well-known text)로부터

```python
print(gPolygon)
```

- WKB(well-known binary)로부터

```python
g = QgsGeometry() wkb = bytes.fromhex("01010000000000000000004540000000000000140") g.fromJson(wkb) # print WKT representation of the geometry print(g.asWkt())
```

### 7.2 도형에 접근

First, you should find out the geometry type. The wkbType() method is the one to use. It returns a value from the QgsWkbTypes.Type enumeration.

```python
if gPnt.wkbType() == QgsWkbTypes.Point:
    print(gPnt.wkbType())
    # output: 1 for Point
if gLine.wkbType() == QgsWkbTypes.LineString:
    print(gLine.wkbType())
    # output: 2 for LineString
if gPolygon.wkbType() == QgsWkbTypes.Polygon:
    print(gPolygon.wkbType())
    # output: 3 for Polygon
```

As an alternative, one can use the type() method which returns a value from the QgsWkbTypes.GeometryType enumeration.

You can use the displayString() function to get a human readable geometry type.

```python
print(QgsWkbTypes.displayString(gPnt.wkbType()))
    # output: 'Point'
print(QgsWkbTypes.displayString(gLine.wkbType()))
    # output: 'LineString'
print(QgsWkbTypes.displayString(gPolygon.wkbType()))
    # output: 'Polygon'
```
There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

To extract information from a geometry there are accessor functions for every vector type. Here’s an example on how to use these accessors:

```python
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)}>]
```

참고: The tuples (x,y) are not real tuples, they are `QgsPoint` objects, the values are accessible with `x()` and `y()` methods.

For multipart geometries there are similar accessor functions: `asMultiPoint()`, `asMultiPolyline()` and `asMultiPolygon()`.

It is possible to iterate over all the parts of a geometry, regardless of the geometry’s type. E.g.

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

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

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

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

It’s also possible to modify each part of the geometry using `QgsGeometry.parts()` method.

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

```python
MultiPoint ((-10334726.79314761981368065 -5360105.10101188533008099),(-10462133.8291775062680244 -5217484.34365727473050356),(-10589398.51346865110099316 -5072020.358805269934237))
```
7.3 도형 관계계산 및 연산

QGIS uses GEOS library for advanced geometry operations such as geometry predicates (contains(), intersects(), ...) and set operations (combine(), difference(), ...). It can also compute geometric properties of geometries, such as area (in the case of polygons) or lengths (for polygons and lines).

Let’s see an example that combines iterating over the features in a given layer and performing some geometric computations based on their geometries. The below code will compute and print the area and perimeter of each country in the countries layer within our tutorial QGIS project.

The following code assumes layer is a QgsVectorLayer object that has Polygon feature type.

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

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

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

Zambia
Area: 62.822790653431205
Perimeter: 50.65232014052552
Zimbabwe
Area: 33.41113559136521
Perimeter: 26.608288555013935

Now you have calculated and printed the areas and perimeters of the geometries. You may however quickly notice that the values are strange. That is because areas and perimeters don’t take CRS into account when computed using the area() and length() methods from the QgsGeometry class. For a more powerful area and distance calculation, the QgsDistanceArea class can be used, which can perform ellipsoid based calculations:

The following code assumes layer is a QgsVectorLayer object that has Polygon feature type.

```
d = QgsDistanceArea()
d.setEllipsoid('WGS84')
layer = QgsProject.instance().mapLayersByName('countries')[0]

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

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

(다음 페이지에 계속)
# 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): 5539361.250080013
Area (m2): 752000605894.2937
Area (km2): 752000.6058942936
Zimbabwe
Perimeter (m): 2865021.3323912495
Area (m2): 389250992553.95465
Area (km2): 389250.99255395465

Alternatively, you may want to know the distance and bearing between two points.

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

# Let's create two points.
# Santa claus is a workaholic and needs a summer break,
# let's 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))

QGIS 에 포함되어 있는, 벡터 데이터를 분석하고 변환하는 데 사용할 수 있는 알고리즘들의 수많은 예시가 있습니다. 다음은 링크들은 그 가운데 몇몇 코드를 보여줍니다.

- Distance and area using the QgsDistanceArea class: Distance matrix algorithm
- Lines to polygons algorithm

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import 
QgsCoordinateReferenceSystem,
QgsCoordinateTransform,
QgsProject,
QgsPointXY,
```
8.1 좌표계

Coordinate reference systems (CRS) are encapsulated by the QgsCoordinateReferenceSystem class. Instances of this class can be created in several different ways:

- 좌표계의 ID 로 CRS 설정

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

QGIS supports different CRS identifiers with the following formats:

- EPSG:<code> — ID assigned by the EPSG organization - handled with createFromOgcWms()
- POSTGIS:<srid> — ID used in PostGIS databases - handled with createFromSrid()
- INTERNAL:<srsid> — ID used in the internal QGIS database - handled with createFromSrsId()
- PROJ:<proj> — handled with createFromProj()
- WKT:<wkt> — handled with createFromWkt()

If no prefix is specified, WKT definition is assumed.

- WKT(well-known text) 로 CRS 설정

  ```python
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257216453653]],'
  'PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],'
  'AXIS["Longitude",EAST], AXIS["Latitude",NORTH])'
  crs = QgsCoordinateReferenceSystem(wkt)
  print(crs.isValid())
  ```
create an invalid CRS and then use one of the create* functions to initialize it. In the following example we use a Proj string to initialize the projection.

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

It’s wise to check whether creation (i.e. lookup in the database) of the CRS has been successful: `isValid()` must return `True`.

Note that for initialization of spatial reference systems QGIS needs to look up appropriate values in its internal database `srs.db`. Thus in case you create an independent application you need to set paths correctly with `QgsApplication.setPrefixPath()`, otherwise it will fail to find the database. If you are running the commands from the QGIS Python console or developing a plugin you do not care: everything is already set up for you.

Accessing spatial reference system information:

```python
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())
print("Is geographic:", crs.isGeographic())
print("Map units:", crs.mapUnits())
```

Output:

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

### 8.2 CRS Transformation

You can do transformation between different spatial reference systems by using the `QgsCoordinateTransform` class. The easiest way to use it is to create a source and destination CRS and construct a `QgsCoordinateTransform` instance with them and the current project. Then just repeatedly call `transform()` function to do the transformation. By default it does forward transformation, but it is capable to do also inverse transformation.

```python
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)

Output:
Transformed point: <QgsPointXY: POINT(832713.79873844375813869 553423.986833143945783)>
Transformed back: <QgsPointXY: POINT(18 5)>

The code snippets on this page need the following imports if you’re outside the qgis console:

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

from qgis.QtCore import Qt, QRectF

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

from qgis.gui import (QgsMapCanvas, QgsVertexMarker, QgsMapCanvasItem, QgsRubberBand, )
```
The Map canvas widget is probably the most important widget within QGIS because it shows the map composed from overlaid map layers and allows interaction with the map and layers. The canvas always shows a part of the map defined by the current canvas extent. The interaction is done through the use of map tools: there are tools for panning, zooming, identifying layers, measuring, vector editing and others. Similar to other graphics programs, there is always one tool active and the user can switch between the available tools.

The map canvas is implemented with the QgsMapCanvas class in the qgis.gui module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please read the overview of the framework.

Whenever the map has been panned, zoomed in/out (or some other action that triggers a refresh), the map is rendered again within the current extent. The layers are rendered to an image (using the QgsMapRendererJob class) and that image is displayed on the canvas. The QgsMapCanvas class also controls refreshing of the rendered map. Besides this item which acts as a background, there may be more map canvas items.

Typical map canvas items are rubber bands (used for measuring, vector editing etc.) or vertex markers. The canvas items are usually used to give visual feedback for map tools, for example, when creating a new polygon, the map tool creates a rubber band canvas item that shows the current shape of the polygon. All map canvas items are subclasses of QgsMapCanvasItem which adds some more functionality to the basic QGraphicsItem objects.

요약하면 캔버스 아키텍처는 다음 3 가지 개념으로 이루어집니다.
• 맵 캔버스—맵을 보여주는 데 쓰입니다.
• map canvas items —additional items that can be displayed on the map canvas
• map tools —for interaction with the map canvas

9.1 맵 캔버스 내장시키기

Map canvas is a widget like any other Qt widget, so using it is as simple as creating and showing it.

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

This produces a standalone window with map canvas. It can be also embedded into an existing widget or window. When using .ui files and Qt Designer, place a QWidget on the form and promote it to a new class: set QgsMapCanvas as class name and set qgis.gui as header file. The pyuic5 utility will take care of it. This is a very convenient way of embedding the canvas. The other possibility is to manually write the code to construct map canvas and other widgets (as children of a main window or dialog) and create a layout.

맵 캔버스의 초기값은 배경은 검은색, 안티알리아싱은 사용하지 않는 것입니다. 배경을 하얀색으로 설정하고 부드러운 렌더링을 위해 안티알리아싱을 활성화하려면 다음과 같이 하십시오.

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

(In case you are wondering, Qt comes from PyQt.QtCore module and Qt.white is one of the predefined QColor instances.)

Now it is time to add some map layers. We will first open a layer and add it to the current project. Then we will set the canvas extent and set the list of layers for the canvas.

```python
vlayer = QgsVectorLayer('testdata/airports.shp', 'Airports layer', 'ogr')
if not vlayer.isValid():
    print("Layer failed to load!")

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

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

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

이 명령어들을 실행하면, 사용자가 불러온 레이어가 캔버스에 보일 것입니다.
9.2 고무줄과 버텍스 마커

To show some additional data on top of the map in canvas, use map canvas items. It is possible to create custom canvas item classes (covered below), however there are two useful canvas item classes for convenience: QgsRubberBand for drawing polylines or polygons, and QgsVertexMarker for drawing points. They both work with map coordinates, so the shape is moved/scaled automatically when the canvas is being panned or zoomed.

To show a polyline:

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

폴리곤을 표시하는 방법은 다음과 같습니다.

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

폴리곤의 포인트들이 1차원 리스트가 아니라는 점에 주의하십시오. 실제로, 폴리곤의 포인트들은 폴리곤의 선형 폐곡선을 담고 있는 폐곡선 리스트입니다. 첫 번째 폐곡선은 외곽 경계선이고, 그 다음의 (있을 수도 있고 없을 수도 있는) 폐곡선은 폴리곤 내부의 구멍에 해당합니다.

고무줄을 사용자 지정 할 수 있습니다. 즉 색상 및 선 두께를 변경할 수도 있습니다.

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

The canvas items are bound to the canvas scene. To temporarily hide them (and show them again), use the hide() and show() combo. To completely remove the item, you have to remove it from the scene of the canvas

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

(C++의 경우 아이템을 그냥 삭제하는 것도 가능하지만, 파이션에서 del r 명령어는 참조만 삭제할 뿐 실제 오브젝트는 캔버스가 소유하고 있으므로 계속 남아 있을 것입니다.)

Rubber band can be also used for drawing points, but the QgsVertexMarker class is better suited for this (QgsRubberBand would only draw a rectangle around the desired point).

You can use the vertex marker like this:

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

This will draw a red cross on position [10,45]. It is possible to customize the icon type, size, color and pen width

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

For temporary hiding of vertex markers and removing them from canvas, use the same methods as for rubber bands.
9.3 캔버스에서 맵 도구 사용하기

The following example constructs a window that contains a map canvas and basic map tools for map panning and zooming. Actions are created for activation of each tool: panning is done with QgsMapToolPan, zooming in/out with a pair of QgsMapToolZoom instances. The actions are set as checkable and later assigned to the tools to allow automatic handling of checked/unchecked state of the actions —when a map tool gets activated, its action is marked as selected and the action of the previous map tool is deselected. The map tools are activated using setMapTool() method.

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

class MyWnd(QMainWindow):
    def __init__(self, layer):
        QMainWindow.__init__(self)
        self.canvas = QgsMapCanvas()
        self.canvas.setCanvasColor(Qt.white)
        self.canvas.setExtent(layer.extent())
        self.canvas.setLayers([layer])
        self.setCentralWidget(self.canvas)
        self.actionZoomIn = QAction("Zoom in", self)
        self.actionZoomOut = QAction("Zoom out", self)
        self.actionPan = QAction("Pan", self)
        self.actionZoomIn.setCheckable(True)
        self.actionZoomOut.setCheckable(True)
        self.actionPan.setCheckable(True)
        self.actionZoomIn.triggered.connect(self.zoomIn)
        self.actionZoomOut.triggered.connect(self.zoomOut)
        self.actionPan.triggered.connect(self.pan)
        self.toolbar = self.addToolBar("Canvas actions")
        self.toolbar.addAction(self.actionZoomIn)
        self.toolbar.addAction(self.actionZoomOut)
        self.toolbar.addAction(self.actionPan)

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

        self.pan()

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

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

(다음 페이지에 계속)
```python
def pan(self):
    self.canvas.setMapTool(self.toolPan)
```

You can try the above code in the Python console editor. To invoke the canvas window, add the following lines to instantiate the MyWnd class. They will render the currently selected layer on the newly created canvas.

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

### 9.3.1 Select a feature using QgsMapToolIdentifyFeature

You can use the map tool `QgsMapToolIdentifyFeature` for asking to the user to select a feature that will be sent to a callback function.

```python
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)
```

### 9.4 사용자 지정 맵 도구 작성

You can write your custom tools, to implement a custom behavior to actions performed by users on the canvas.

Map tools should inherit from the `QgsMapTool`, class or any derived class, and selected as active tools in the canvas using the `setMapTool()` method as we have already seen.

다음 예시 코드는 캔버스 상에서 클릭과 드래그로 사각형 범위를 정의하도록 해주는 맵 도구입니다. 사각형이 정의되면, 콘솔에 그 범위 좌표를 출력합니다. 이전에 설명했던 고무줄 기능을 사용해서 확정되기 전의 사각형을 표시할 것입니다.

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

    def reset(self):
        self startPoint = self endPoint = None
```

(다음 페이지에 계속)
self.isEmittingPoint = False
self.rubberBand.reset(True)

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

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

def canvasMoveEvent(self, e):
    if not self.isEmittingPoint:
        return
    self.endPoint = self.toMapCoordinates(e.pos())
    self.showRect(self.startPoint, self.endPoint)

def showRect(self, startPoint, endPoint):
    self.rubberBand.reset(QGis.Polygon)
    if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
        return
    point1 = QgsPoint(startPoint.x(), startPoint.y())
    point2 = QgsPoint(startPoint.x(), endPoint.y())
    point3 = QgsPoint(endPoint.x(), endPoint.y())
    point4 = QgsPoint(endPoint.x(), startPoint.y())
    self.rubberBand.addPoint(point1, False)
    self.rubberBand.addPoint(point2, False)
    self.rubberBand.addPoint(point3, False)
    self.rubberBand.addPoint(point4, True)  # true to update canvas
    self.rubberBand.show()

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

def deactivate(self):
    QgsMapTool.deactivate(self)
    self.deactivated.emit()
9.5 사용자 지정 맵 캔버스 아이템 작성

Here is an example of a custom canvas item that draws a circle:

```python
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)
```

The code snippets on this page need the following imports:

```python
import os
from qgis.core import (QgsGeometry,
                        QgsMapSettings,
                        QgsPrintLayout,
                        QgsMapSettings,
                        QgsMapRendererParallelJob,
                        QgsLayoutItemLabel,
                        QgsLayoutItemLegend,
                        QgsLayoutItemMap,
                        QgsLayoutItemPolygon,
                        QgsLayoutItemScaleBar,
                        QgsLayoutExporter,
                        QgsLayoutItem,
```

(다음 페이지에 계속)
QgsLayoutPoint,
QgsLayoutSize,
QgsUnitTypes,
QgsProject,
QgsFillSymbol,

from qgis.PyQt.QtGui import
    QPolygonF,
    QColor,

from qgis.PyQt.QtCore import
    QPointF,
    QRectF,
    QSize,
There are generally two approaches when input data should be rendered as a map: either do it quick way using `QgsMapRendererJob` or produce more fine-tuned output by composing the map with the `QgsLayout` class.

### 10.1 단순 렌더링

The rendering is done creating a `QgsMapSettings` object to define the rendering settings, and then constructing a `QgsMapRendererJob` with those settings. The latter is then used to create the resulting image.

```python
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)
(다음 페이지에 계속)```
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 서로 다른 좌표계를 가진 레이어들을 렌더링하기

If you have more than one layer and they have a different CRS, the simple example above will probably not work: to get the right values from the extent calculations you have to explicitly set the destination CRS

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

10.3 Output using print layout

Print layout is a very handy tool if you would like to do a more sophisticated output than the simple rendering shown above. It is possible to create complex map layouts consisting of map views, labels, legend, tables and other elements that are usually present on paper maps. The layouts can be then exported to PDF, raster images or directly printed on a printer.

The layout consists of a bunch of classes. They all belong to the core library. QGIS application has a convenient GUI for placement of the elements, though it is not available in the GUI library. If you are not familiar with Qt Graphics View framework, then you are encouraged to check the documentation now, because the layout is based on it.

The central class of the layout is the QgsLayout class, which is derived from the Qt QGraphicsScene class. Let us create an instance of it:

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

This initializes the layout with some default settings, specifically by adding an empty A4 page to the layout. You can create layouts without calling the initializeDefaults() method, but you’ll need to take care of adding pages to the layout yourself.

The previous code creates a temporary layout that is not visible in the GUI. It can be handy to e.g. quickly add some items and export without modifying the project itself nor expose these changes to the user. If you want the layout to be saved/restored along with the project and available in the layout manager, then add:
Now we can add various elements (map, label, ...) to the layout. All these objects are represented by classes that inherit from the base `QgsLayoutItem` class.

Here’s a description of some of the main layout items that can be added to a layout.

- **map** — Here we create a map of a custom size and render the current map canvas

  ```python
  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)
  ```

- **라벨** — 라벨을 표출할 수 있도록 해줍니다. 폰트, 색상, 정렬 및 여백을 수정할 수 있습니다.

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

- **범례**

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

- **축척막대 (scale bar)**

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

- **화살표**
- **그림**
- **basic shape**
- **nodes based shape**

  ```python
  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)
  ```

  ```python
  props = {}
  props["color"] = "green"
  ```

  (다음 페이지에 계속)
• 포

Once an item is added to the layout, it can be moved and resized:

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

A frame is drawn around each item by default. You can remove it as follows:

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

Besides creating the layout items by hand, QGIS has support for layout templates which are essentially compositions with all their items saved to a .qpt file (with XML syntax).

Once the composition is ready (the layout items have been created and added to the composition), we can proceed to produce a raster and/or vector output.

### 10.3.1 Exporting the layout

To export a layout, the `QgsLayoutExporter` class must be used.

```python
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())
```

Use the `exportToImage()` in case you want to export to an image instead of a PDF file.

### 10.3.2 Exporting a layout atlas

If you want to export all pages from a layout that has the atlas option configured and enabled, you need to use the `atlas()` method in the exporter (`QgsLayoutExporter`) with small adjustments. In the following example, the pages are exported to PNG images:

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

Notice that the outputs will be saved in the base path folder, using the output filename expression configured on atlas.

The code snippets on this page need the following imports if you’re outside the pyqgis console:
```python
from qgis.core import (
    edit,
    QgsExpression,
    QgsExpressionContext,
    QgsFeature,
    QgsFeatureRequest,
    QgsField,
    QgsFields,
    QgsVectorLayer,
    QgsPointXY,
    QgsGeometry,
    QgsProject,
    QgsExpressionContextUtils
)
```
CHAPTER 11

표현식을 이용한 속성값의 필터링 및 계산

• 표현식 파싱하기
• 표현식 평가하기
  – 기본 표현식
  – 피처와 표현식
  – Filtering a layer with expressions
• Handling expression errors

QGIS has some support for parsing of SQL-like expressions. Only a small subset of SQL syntax is supported. The expressions can be evaluated either as boolean predicates (returning True or False) or as functions (returning a scalar value). See vector_expressions in the User Manual for a complete list of available functions.

d음 3 가지 기본 유형이 지원됩니다.
  • 숫자—정수와 십진수, 예를 들면 123, 3.14
  • 문자열—'hello world'처럼 작은 따옴표로 둘러싸야 함
  • 컬럼 참조—표현식을 평가할 때, 참조가 필드의 실제 값으로 대체됩니다. 이름이 그대로 남지 않습니다.
다음과 같은 연산자들이 있습니다:
  • 산술 연산자: +, −, *, /, ^
  • 괄호: (1 + 1) * 3처럼 연산의 우선 순위를 강제합니다.
  • 단항 플러스 및 마이너스: -12, +5
  • 수학 함수: sqrt, sin, cos, tan, asin, acos, atan
  • conversion functions: to_int, to_real, to_string, to_date
  • 도형 함수: $area, $length
  • geometry handling functions: $x, $y, $geometry, num_geometries, centroid
다음과 같은 서술어들을 지원합니다.

- 비교: =, !=, >, >=, <, <=
- 패턴 매칭: LIKE ($와 _ 사용), ~ (정규 표현식)
- 논리 술어: AND, OR, NOT
- NULL 값 확인: IS NULL, IS NOT NULL

서술어의 예:

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

스칼라 표현식의 예:

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

### 11.1 표현식 파싱하기

The following example shows how to check if a given expression can be parsed correctly:

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

### 11.2 표현식 평가하기

Expressions can be used in different contexts, for example to filter features or to compute new field values. In any case, the expression has to be evaluated. That means that its value is computed by performing the specified computational steps, which can range from simple arithmetic to aggregate expressions.

### 11.2.1 기본 표현식

This basic expression evaluates a simple arithmetic operation:

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

6
Expression can also be used for comparison, evaluating to 1 (True) or 0 (False)

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

### 11.2.2 피처와 표현식

To evaluate an expression against a feature, a `QgsExpressionContext` object has to be created and passed to the `evaluate` function in order to allow the expression to access the feature’s field values.

The following example shows how to create a feature with a field called《Column》and how to add this feature to the expression context.

```python
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
```

The following is a more complete example of how to use expressions in the context of a vector layer, in order to compute new field values:

```python
from qgis.PyQt.QtCore import QVariant

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

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

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

(다음 페이지에 계속)
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 Filtering a layer with expressions

다음 예시 코드는 레이어를 필터링해서 서술어와 일치하는 모든 피처를 반환하는 작업에 사용할 수 있습니다.

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

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

expression = 'Test >= 3'
request = QgsFeatureRequest().setFilterExpression(expression)
matches = 0
for f in layer.getFeatures(request):
    matches += 1

print(matches)
11.3 Handling expression errors

Expression-related errors can occur during expression parsing or evaluation:

```python
exp = QgsExpression("1 + 1 = 2")
if exp.hasParserError():
    raise Exception(exp.parserErrorString())
value = exp.evaluate()
if exp.hasEvalError():
    raise ValueError(exp.evalErrorString())
```

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import (QgsProject,
                       QgsSettings,
                       QgsVectorLayer
)
```
표현식을 이용한 속성값의 필터링 및 계산
Many times it is useful for a plugin to save some variables so that the user does not have to enter or select them again next time the plugin is run.

These variables can be saved and retrieved with help of Qt and QGIS API. For each variable, you should pick a key that will be used to access the variable—for user’s favourite color you could use key ‘favourite_color’ or any other meaningful string. It is recommended to give some structure to naming of keys.

We can differentiate between several types of settings:

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

Here you can see an example of how these methods are used.

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

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

The second parameter of the value() method is optional and specifies the default value that is returned if there is no previous value set for the passed setting name.
For a method to pre-configure the default values of the global settings through the `global_settings.ini` file, see deploying_organization for further details.

- **project settings** — vary between different projects and therefore they are connected with a project file. Map canvas background color or destination coordinate reference system (CRS) are examples — white background and WGS84 might be suitable for one project, while yellow background and UTM projection are better for another one.

An example of usage follows.

```python
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)
```

As you can see, the `writeEntry()` method is used for many data types (integer, string, list), but several methods exist for reading the setting value back, and the corresponding one has to be selected for each data type.

- **map layer settings** — these settings are related to a particular instance of a map layer with a project. They are not connected with underlying data source of a layer, so if you create two map layer instances of one shapefile, they will not share the settings. The settings are stored inside the project file, so if the user opens the project again, the layer-related settings will be there again. The value for a given setting is retrieved using the `customProperty()` method, and can be set using the `setCustomProperty()` one.

```python
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")
```

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import
   QgsMessageLog,
   QgsGeometry,
)
```
from qgis.gui import ( QgsMessageBar, )
from qgis.PyQt.QtWidgets import ( QSizePolicy, QPushButton, QDialog, QGridLayout, QDialogButtonBox, )
사용자에게 정보 전달하기

이 장에서는 사용자 인터페이스에서 일관성을 유지하기 위해 이용해야 하는 몇몇 메소드 및 요소들에 대해 설명합니다.

13.1 Showing messages. The QgsMessageBar class

메시지 상자를 이용하는 것은 사용자 경험이라는 관점에서 그리 좋은 생각이 아닙니다. 짧은 정보 또는 경고/오류 메시지를 표출하는 데에는 QGIS 메시지 바를 이용하는 편이 더 낫습니다.

QGIS 인터페이스 오브젝트의 참조 (reference) 를 사용하면, 다음 코드를 통해 메시지 바에 메시지를 표시할 수 있습니다.

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

지속 시간 (duration) 을 설정해서 제한된 시간 동안 메시지를 표출할 수 있습니다.

```python
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

The examples above show an error bar, but the level parameter can be used to creating warning messages or info messages, using the Qgis.MessageLevel enumeration. You can use up to 4 different levels:

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

Messages(1): Missing Layers : Show Me

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

별도의 메시지 상자가 필요 없거나 QGIS 메인 창에 메시지를 표시할 이유가 없을 경우, 사용자의 대화창에 메시지 바를 표시할 수도 있습니다.
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()

13.2 진행률 표시

앞에서 배웠듯이 QGIS 메시지 바에 위젯을 추가할 수 있으므로, 진행률 (progress) 막대도 추가할 수 있습니다. 다음은 여러분이 콘솔에서 실행해볼 수 있는 예시 코드입니다.

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(i)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()

Messages(0): Doing something boring...

Also, you can use the built-in status bar to report progress, as in the next example:

vlayer = iface.activeLayer()
그림 13.5: 사용자 지정 대화창 내부의 QGIS 메시지 바
```python
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 {0} %").format(int(percent))
    iface.statusBarIface().showMessage("Processed {0} %".format(int(percent)))

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

### 13.3 로그 기록

There are three different types of logging available in QGIS to log and save all the information about the execution of your code. Each has its specific output location. Please consider to use the correct way of logging for your purpose:

- **QgsMessageLog** is for messages to communicate issues to the user. The output of the QgsMessageLog is shown in the Log Messages Panel.
- The python built in [logging](https://docs.python.org/3/library/logging.html) module is for debugging on the level of the QGIS Python API (PyQGIS). It is recommended for Python script developers that need to debug their python code, e.g. feature ids or geometries.
- **QgsLogger** is for messages for QGIS internal debugging / developers (i.e. you suspect something is triggered by some broken code). Messages are only visible with developer versions of QGIS.

Examples for the different logging types are shown in the following sections below.

경고: Use of the Python **print** statement is unsafe to do in any code which may be multithreaded and extremely slows down the algorithm. This includes [expression functions](https://qgis.org/api/python-qgis-3.16.html#qgis.PyQGIS_Expression), [renderers](https://qgis.org/api/python-qgis-3.16.html#qgis.PyQGIS_Renderers), [symbol layers](https://qgis.org/api/python-qgis-3.16.html#qgis.PyQGIS_SymbolLayers) and [Processing algorithms](https://qgis.org/api/python-qgis-3.16.html) (amongst others). In these cases you should always use the python logging module or thread safe classes ([QgsLogger](https://qgis.org/api/python-qgis-3.16.html#qgis.PyQGIS_Logger) or [QgsMessageLog](https://qgis.org/api/python-qgis-3.16.html#qgis.PyQGIS_MessageLog)) instead.

#### 13.3.1 QgsMessageLog

```python
# 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!

참고: You can see the output of the QgsMessageLog in the log_message_panel
13.3.2 The python built in logging module

```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")
```

The `basicConfig` method configures the basic setup of the logging. In the above code the filename, logging level and the format are defined. The filename refers to where to write the logfile to, the logging level defines what levels to output and the format defines the format in which each message is output.

If you want to erase the log file every time you execute your script you can do something like:

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

Further resources on how to use the python logging facility are available at:

- 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

경고: Please note that without logging to a file by setting a filename the logging may be multithreaded which heavily slows down the output.

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import {
    QgsApplication,
    QgsRasterLayer,
    QgsAuthMethodConfig,
    QgsDataSourceUri,
    QgsPkiBundle,
    QgsMessageLog,
}

from qgis.gui import {
    QgsAuthAuthoritiesEditor,
    QgsAuthConfigEditor,
    QgsAuthConfigSelect,
    QgsAuthSettingsWidget,
}

from qgis.QtWidgets import {
    QWidget,
    QTabWidget,
}
```

(다음 페이지에 계속)
from qgis.PyQt.QtNetwork import QSslCertificate
Authentication infrastructure

- 소개
- Glossary
- QgsAuthManager the entry point
  - Init the manager and set the master password
  - Populate authdb with a new Authentication Configuration entry
    * Available Authentication methods
    * Populate Authorities
    * Manage PKI bundles with QgsPkiBundle
  - Remove an entry from authdb
  - Leave authcfg expansion to QgsAuthManager
    * PKI examples with other data providers
- Adapt plugins to use Authentication infrastructure
- Authentication GUIs
  - GUI to select credentials
  - Authentication Editor GUI
  - Authorities Editor GUI
14.1 소개

User reference of the Authentication infrastructure can be read in the User Manual in the authentication_overview paragraph.

This chapter describes the best practices to use the Authentication system from a developer perspective.

The authentication system is widely used in QGIS Desktop by data providers whenever credentials are required to access a particular resource, for example when a layer establishes a connection to a Postgres database.

There are also a few widgets in the QGIS gui library that plugin developers can use to easily integrate the authentication infrastructure into their code:

- QgsAuthConfigEditor
- QgsAuthConfigSelect
- QgsAuthSettingsWidget

A good code reference can be read from the authentication infrastructure tests code.

경고: Due to the security constraints that were taken into account during the authentication infrastructure design, only a selected subset of the internal methods are exposed to Python.

14.2 Glossary

Here are some definition of the most common objects treated in this chapter.

Master Password Password to allow access and decrypt credential stored in the QGIS Authentication DB

Authentication Database A Master Password encrypted sqlite db qgis-auth.db where Authentication Configuration are stored. e.g user/password, personal certificates and keys, Certificate Authorities

Authentication DB Authentication Database

인증 환경 설정 A set of authentication data depending on Authentication Method. e.g Basic authentication method stores the couple of user/password.

Authentication Config Authentication Configuration

Authentication Method A specific method used to get authenticated. Each method has its own protocol used to gain the authenticated level. Each method is implemented as shared library loaded dynamically during QGIS authentication infrastructure init.

14.3 QgsAuthManager the entry point

The QgsAuthManager singleton is the entry point to use the credentials stored in the QGIS encrypted Authentication DB, i.e. the qgis-auth.db file under the active user profile folder.

This class takes care of the user interaction: by asking to set a master password or by transparently using it to access encrypted stored information.
14.3.1 Init the manager and set the master password

The following snippet gives an example to set master password to open the access to the authentication settings. Code comments are important to understand the snippet.

```python
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():
    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 Populate authdb with a new Authentication Configuration entry

Any stored credential is a Authentication Configuration instance of the QgsAuthMethodConfig class accessed using a unique string like the following one:

```python
authcfg = 'fm1s770'
```

that string is generated automatically when creating an entry using the QGIS API or GUI, but it might be useful to manually set it to a known value in case the configuration must be shared (with different credentials) between multiple users within an organization.

QgsAuthMethodConfig is the base class for any Authentication Method. Any Authentication Method sets a configuration hash map where authentication information will be stored. Hereafter a useful snippet to store PKI-path credentials for a hypothetical alice user:

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

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

---

Available Authentication methods

**Authentication Method** libraries are loaded dynamically during authentication manager init. Available authentication methods are:

1. Basic User and password authentication
2. Esri-Token ESRI token based authentication
3. Identity-Cert Identity certificate authentication
4. OAuth2 OAuth2 authentication
5. PKI-Paths PKI paths authentication
6. PKI-PKCS#12 PKI PKCS#12 authentication

Populate Authorities

```python
authMgr = QgsApplication.authManager()
# add authorities
cacerts = QSslCertificate.fromPath("/path/to/ca_chains.pem")
assert cacerts is not None
# store CA
authMgr.storeCert Authorities(cacerts)
# and rebuild CA caches
authMgr.rebuildCaCertsCache()
authMgr.rebuildTrustedCaCertsCache()
```

Manage PKI bundles with QgsPkiBundle

A convenience class to pack PKI bundles composed on SslCert, SslKey and CA chain is the `QgsPkiBundle` class. Hereafter a snippet to get password protected:

```python
# 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()
```

Refer to `QgsPkiBundle` class documentation to extract cert/key/CAs from the bundle.

---

Chapter 14. Authentication infrastructure
14.3.3 Remove an entry from authdb

We can remove an entry from Authentication Database using its authcfg identifier with the following snippet:

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

14.3.4 Leave authcfg expansion to QgsAuthManager

The best way to use an Authentication Config stored in the Authentication DB is referring it with the unique identifier authcfg. Expanding, means convert it from an identifier to a complete set of credentials. The best practice to use stored Authentication Configs, is to leave it managed automatically by the Authentication manager. The common use of a stored configuration is to connect to an authentication enabled service like a WMS or WFS or to a DB connection.

참고: Take into account that not all QGIS data providers are integrated with the Authentication infrastructure. Each authentication method, derived from the base class QgsAuthMethod and support a different set of Providers. For example the certIdentity() method supports the following list of providers:

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

Sample output:

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

For example, to access a WMS service using stored credentials identified with authcfg = 'fm1s770', we just have to use the authcfg in the data source URL like in the following snippet:

```python
authCfg = 'fm1s770'
qu = QgsDataSourceUri()
qu.setParam("layers", 'usa:states')
qu.setParam("styles", '')
qu.setParam("format", 'image/png')
qu.setParam("crs", 'EPSG:4326')
qu.setParam("dpiMode", 7)
qu.setParam("featureCount", 10)
qu.setParam("authcfg", authCfg) # ----> here my authcfg url parameter
qu.setParam("contextualWMSLegend", 0)
qu.setParam("url", 'https://my_auth_enabled_server_ip/wms')
rlayer = QgsRasterLayer(str(qu.encodedUri(), "utf-8"), 'states', 'wms')
```

In the upper case, the wms provider will take care to expand authcfg URI parameter with credential just before setting the HTTP connection.

경고: The developer would have to leave authcfg expansion to the QgsAuthManager, in this way he will be sure that expansion is not done too early.

Usually an URI string, built using the QgsDataSourceURI class, is used to set a data source in the following way:

```python
authCfg = 'fm1s770'
qu = QgsDataSourceUri("my WMS uri here")
```
PyQGIS 3.16 developer cookbook

$quri$.setParam("authcfg", authCfg)

$raster = QgsRasterLayer( quri.uri(False), 'states', 'wms')$

참고: The False parameter is important to avoid URI complete expansion of the authcfg id present in the URI.

PKI examples with other data providers

Other example can be read directly in the QGIS tests upstream as in test_authmanager_pki_ows or test_authmanager_pki_postgres.

14.4 Adapt plugins to use Authentication infrastructure

Many third party plugins are using httplib2 or other Python networking libraries to manage HTTP connections instead of integrating with QgsNetworkAccessManager and its related Authentication Infrastructure integration.

To facilitate this integration a helper Python function has been created called NetworkAccessManager. Its code can be found here.

This helper class can be used as in the following snippet:

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

14.5 Authentication GUIs

In this paragraph are listed the available GUIs useful to integrate authentication infrastructure in custom interfaces.

14.5.1 GUI to select credentials

If it’s necessary to select a Authentication Configuration from the set stored in the Authentication DB it is available in the GUI class QgsAuthConfigSelect.

and can be used as in the following snippet:

```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
parent = QWidget() # Your GUI parent widget
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
# GUI has to be integrated
tabGui = QTabWidget()
tabGui.insertTab( 1, gui, "Configurations" )
```
The above example is taken from the QGIS source code. The second parameter of the GUI constructor refers to data provider type. The parameter is used to restrict the compatible Authentication Methods with the specified provider.

### 14.5.2 Authentication Editor GUI

The complete GUI used to manage credentials, authorities and to access to Authentication utilities is managed by the QgsAuthEditorWidgets class.

```
and can be used as in the following snippet:
```
# create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
# the widget referred with ‘parent’
parent = QWidget()  # Your GUI parent widget
gui = QgsAuthConfigSelect( parent )
gui.show()

An integrated example can be found in the related test.

### 14.5.3 Authorities Editor GUI

A GUI used to manage only authorities is managed by the `QgsAuthAuthoritiesEditor` class.

![](image)

and can be used as in the following snippet:

```python
# create the instance of the QgsAuthAuthoritiesEditor GUI hierarchically
# linked to the widget referred with 'parent'
parent = QWidget()  # Your GUI parent widget
```
The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import (QgsProcessingContext,
                       QgsTaskManager,
                       QgsTask,
                       QgsProcessingAlgRunnerTask,
                       Qgis,
                       QgsProcessingFeedback,
                       QgsApplication,
                       QgsMessageLog,
)
```
CHAPTER 15

Tasks - doing heavy work in the background

15.1 소개

Background processing using threads is a way to maintain a responsive user interface when heavy processing is going on. Tasks can be used to achieve threading in QGIS.

A task (`QgsTask`) is a container for the code to be performed in the background, and the task manager (`QgsTaskManager`) is used to control the running of the tasks. These classes simplify background processing in QGIS by providing mechanisms for signaling, progress reporting and access to the status for background processes. Tasks can be grouped using subtasks.

The global task manager (found with `QgsApplication.taskManager()`) is normally used. This means that your tasks may not be the only tasks that are controlled by the task manager.

There are several ways to create a QGIS task:

- Create your own task by extending `QgsTask`

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

- Create a task from a function

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

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

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

- Create a task from a processing algorithm
params = dict()
context = QgsProcessingContext()
feedback = QgsProcessingFeedback()

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

경고: Any background task (regardless of how it is created) must NEVER use any QObject that lives on the main thread, such as accessing QgsVectorLayer, QgsProject or perform any GUI based operations like creating new widgets or interacting with existing widgets. Qt widgets must only be accessed or modified from the main thread. Data that is used in a task must be copied before the task is started. Attempting to use them from background threads will result in crashes.

Dependencies between tasks can be described using the addSubTask() function of QgsTask. When a dependency is stated, the task manager will automatically determine how these dependencies will be executed. Wherever possible dependencies will be executed in parallel in order to satisfy them as quickly as possible. If a task on which another task depends is canceled, the dependent task will also be canceled. Circular dependencies can make deadlocks possible, so be careful.

If a task depends on a layer being available, this can be stated using the setDependentLayers() function of QgsTask. If a layer on which a task depends is not available, the task will be canceled.

Once the task has been created it can be scheduled for running using the addTask() function of the task manager. Adding a task to the manager automatically transfers ownership of that task to the manager, and the manager will cleanup and delete tasks after they have executed. The scheduling of the tasks is influenced by the task priority, which is set in addTask().

The status of tasks can be monitored using QgsTask and QgsTaskManager signals and functions.

### 15.2 예제

#### 15.2.1 Extending QgsTask

In this example `RandomIntegerSumTask` extends QgsTask and will generate 100 random integers between 0 and 500 during a specified period of time. If the random number is 42, the task is aborted and an exception is raised. Several instances of `RandomIntegerSumTask` (with subtasks) are generated and added to the task manager, demonstrating two types of dependencies.

```python
import random
from time import sleep

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

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    
    """This shows how to subclass QgsTask""
```
```python
def __init__(self, description, duration):
    super().__init__(description, QgsTask.CanCancel)
    self.duration = duration
    self.total = 0
    self.iterations = 0
    self.exception = None

def run(self):
    """Here you implement your heavy lifting.
    Should periodically test for isCanceled() to gracefully abort.
    This method MUST return True or False.
    Raising exceptions will crash QGIS, so we handle them internally and raise them in self.finished
    """
    QgsMessageLog.logMessage('Started task "{}"'.format(
        self.description()),
        MESSAGE_CATEGORY, Qgis.Info)

    wait_time = self.duration / 100
    for i in range(100):
        sleep(wait_time)
        # use setProgress to report progress
        self.setProgress(i)
        arandominteger = random.randint(0, 500)
        self.total += arandominteger
        self.iterations += 1
        # check isCanceled() to handle cancellation
        if self.isCanceled():
            return False
        # simulate exceptions to show how to abort task
        if arandominteger == 42:
            # DO NOT raise Exception('bad value!')
            # this would crash QGIS
            self.exception = Exception('bad value!')
            return False
    return True

def finished(self, result):
    """
    This function is automatically called when the task has completed (successfully or not).
    You implement finished() to do whatever follow-up stuff should happen after the task is complete.
    finished is always called from the main thread, so it's safe to do GUI operations and raise Python exceptions here.
    result is the return value from self.run.
    """
    if result:
        QgsMessageLog.logMessage(
            'RandomTask "{}" completed
            RandomTotal: {} (with {} iterations)
            iterations',
            name=self.description(),
            total=self.total,
            iterations=self.iterations),
            MESSAGE_CATEGORY, Qgis.Success)
```

(이전 페이지에서 계속)
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()),
        MESSAGECATEGORY, 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)
RandomTotal: 22534 (with 100 iterations)

15.2.2 Task from function

Create a task from a function (doSomething in this example). The first parameter of the function will hold the
QgsTask for the function. An important (named) parameter is on_finished, that specifies a function that will be
called when the task has completed. The doSomething function in this example has an additional named parameter
wait_time.

```python
import random
from time import sleep

MESSAGECATEGORY = '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()),
                          MESSAGECATEGORY, 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()),
                          MESSAGECATEGORY, 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 \n(probably manually canceled by the user)',
                               MESSAGECATEGORY, Qgis.Warning)
    else:
        QgsMessageLog.logMessage('Task \nTotal: {total} (with {iterations}) \nname=task, 
   total=total, 
   iterations=iterations)',
                               MESSAGECATEGORY, Qgis.Info)
else:
    QgsMessageLog.logMessage('Exception: {}'.format(exception),
                               MESSAGECATEGORY, Qgis.Critical)
raise exception

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

15.2.3 Task from a processing algorithm

Create a task that uses the algorithm qgis:randompointsinextent to generate 50000 random points inside a specified extent. The result is added to the project in a safe way.

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

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

See also: https://opengis.ch/2018/06/22/threads-in-pyqgis3/.
16.1 Structuring Python Plugins

- Writing a plugin
  - Plugin files
- Plugin content
  - Plugin metadata
  - __init__.py
  - mainPlugin.py
  - Resource File
- Documentation
- Translation
  - Software requirements
  - Files and directory
    * .pro file
    * .ts file
    * .qm file
  - Translate using Makefile
  - Load the plugin
- Tips and Tricks
  - Plugin Reloader
In order to create a plugin, here are some steps to follow:

1. **Idea**: Have an idea about what you want to do with your new QGIS plugin. Why do you do it? What problem do you want to solve? Is there already another plugin for that problem?

2. **Create files**: some are essentials (see Plugin files)

3. **Write code**: Write the code in appropriate files

4. **Test**: Reload your plugin to check if everything is OK

5. **Publish**: Publish your plugin in QGIS repository or make your own repository as an arsenal of personal GIS weapons.

### 16.1.1 Writing a plugin

Since the introduction of Python plugins in QGIS, a number of plugins have appeared. The QGIS team maintains an [Official Python plugin repository](https://plugins.qgis.org). You can use their source to learn more about programming with PyQGIS or find out whether you are duplicating development effort.

#### Plugin files

Here’s the directory structure of our example plugin

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

What is the meaning of the files:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.py` = The main working code of the plugin. Contains all the information about the actions of the plugin and the main code.
- `resources.qrc` = The `.xml` document created by Qt Designer. Contains relative paths to resources of the forms.
- `resources.py` = The translation of the `.qrc` file described above to Python.
- `form.ui` = The GUI created by Qt Designer.
- `form.py` = The translation of the `form.ui` described above to Python.
- `metadata.txt` = Contains general info, version, name and some other metadata used by plugins website and plugin infrastructure.
Here is a way of creating the basic files (skeleton) of a typical QGIS Python plugin.

There is a QGIS plugin called Plugin Builder 3 that creates a plugin template for QGIS. This is the recommended option, as it produces 3.x compatible sources.

경고: If you plan to upload the plugin to the Official Python plugin repository you must check that your plugin follows some additional rules, required for plugin Validation

16.1.2 Plugin content

Here you can find information and examples about what to add in each of the files in the file structure described above.

Plugin metadata

First, the plugin manager needs to retrieve some basic information about the plugin such as its name, description etc. File metadata.txt is the right place to put this information.

참고: All metadata must be in UTF-8 encoding.

<table>
<thead>
<tr>
<th>Metadata name</th>
<th>필수 여부</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>True</td>
<td>a short string containing the name of the plugin</td>
</tr>
<tr>
<td>qgisMinimumVersion</td>
<td>True</td>
<td>dotted notation of minimum QGIS version</td>
</tr>
<tr>
<td>qgisMaximumVersion</td>
<td>False</td>
<td>dotted notation of maximum QGIS version</td>
</tr>
<tr>
<td>설명</td>
<td>True</td>
<td>short text which describes the plugin, no HTML allowed</td>
</tr>
<tr>
<td>about</td>
<td>True</td>
<td>longer text which describes the plugin in details, no HTML allowed</td>
</tr>
<tr>
<td>version</td>
<td>True</td>
<td>short string with the version dotted notation</td>
</tr>
<tr>
<td>author</td>
<td>True</td>
<td>author name</td>
</tr>
<tr>
<td>email</td>
<td>True</td>
<td>email of the author, only shown on the website to logged in users, but visible in the Plugin Manager after the plugin is installed</td>
</tr>
<tr>
<td>changelog</td>
<td>False</td>
<td>string, can be multilime, no HTML allowed</td>
</tr>
<tr>
<td>experimental</td>
<td>False</td>
<td>boolean flag, True or False - True if this version is experimental</td>
</tr>
<tr>
<td>deprecated</td>
<td>False</td>
<td>boolean flag, True or False, applies to the whole plugin and not just to the uploaded version</td>
</tr>
<tr>
<td>태그</td>
<td>False</td>
<td>comma separated list, spaces are allowed inside individual tags</td>
</tr>
<tr>
<td>homepage</td>
<td>False</td>
<td>a valid URL pointing to the homepage of your plugin</td>
</tr>
<tr>
<td>repository</td>
<td>True</td>
<td>a valid URL for the source code repository</td>
</tr>
<tr>
<td>tracker</td>
<td>False</td>
<td>a valid URL for tickets and bug reports</td>
</tr>
<tr>
<td>icon</td>
<td>False</td>
<td>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)</td>
</tr>
<tr>
<td>category</td>
<td>False</td>
<td>one of Raster, Vector, Database and Web</td>
</tr>
<tr>
<td>plugin_dependencies</td>
<td>False</td>
<td>PIP-like comma separated list of other plugins to install</td>
</tr>
<tr>
<td>server</td>
<td>False</td>
<td>boolean flag, True or False, determines if the plugin has a server interface</td>
</tr>
<tr>
<td>hasProcessing-Provider</td>
<td>False</td>
<td>boolean flag, True or False, determines if the plugin provides processing algorithms</td>
</tr>
</tbody>
</table>
By default, plugins are placed in the *Plugins* menu (we will see in the next section how to add a menu entry for your plugin) but they can also be placed into *Raster*, *Vector*, *Database* and *Web* menus.

A corresponding 《category》 metadata entry exists to specify that, so the plugin can be classified accordingly. This metadata entry is used as tip for users and tells them where (in which menu) the plugin can be found. Allowed values for 《category》 are: Vector, Raster, Database or Web. For example, if your plugin will be available from *Raster* menu, add this to *metadata.txt*

```plaintext
category=Raster
```

참고: If *gisMaximumVersion* is empty, it will be automatically set to the major version plus .99 when uploaded to the *Official Python plugin repository*.

An example for this metadata.txt

```plaintext
; the next section is mandatory

[general]
name=HelloWorld
e-mail=me@example.com
author=Just Me
ggisMinimumVersion=3.0
description=This *is* an example plugin *for* greeting the world.
   Multiline *is* allowed:
      lines starting *with* spaces belong to the same
      field, *in* this case to the "description" field.
      HTML formatting *is* not allowed.
about=This paragraph can contain a detailed description
   of the plugin. Multiline *is* allowed, HTML *is not*.
version=version 1.2
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo

; 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)
```

(다음 페이지에 계속)
deprecated=True

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"
plugin_dependencies=MyOtherPlugin==1.12,YetAnotherPlugin

__init__.py

This file is required by Python’s import system. Also, QGIS requires that this file contains a classFactory() function,
which is called when the plugin gets loaded into QGIS. It receives a reference to the instance of QgisInterface and
must return an object of your plugin’s class from the mainplugin.py — in our case it’s called TestPlugin (see
below). This is how __init__.py should look like

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

# any other initialisation needed

mainPlugin.py

This is where the magic happens and this is how magic looks like: (e.g. mainPlugin.py)

from qgis.QtGui import *
from qgis.QtWidgets import *

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

class TestPlugin:

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

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

        # add toolbar button and menu item
        self.iface.addToolBarIcon(self.action)
        self.iface.addPluginToMenu("&Test plugins", self.action)
# connect to signal renderComplete which is emitted when canvas rendering is done
self.iface.mapCanvas().renderComplete.connect(self.renderTest)

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

    # disconnect form signal of the canvas
    self.iface.mapCanvas().renderComplete.disconnect(self.renderTest)

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

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

The only plugin functions that must exist in the main plugin source file (e.g. mainPlugin.py) are:

- `__init__` which gives access to QGIS interface
- `initGui()` called when the plugin is loaded
- `unload()` called when the plugin is unloaded

In the above example, `addPluginToMenu()` is used. This will add the corresponding menu action to the `Plugins` menu. Alternative methods exist to add the action to a different menu. Here is a list of those methods:

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

All of them have the same syntax as the `addPluginToMenu()` method.

Adding your plugin menu to one of those predefined method is recommended to keep consistency in how plugin entries are organized. However, you can add your custom menu group directly to the menu bar, as the next example demonstrates:

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

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

    menuBar = self.iface.mainWindow().menuBar()
Don’t forget to set QAction and QMenu objectName to a name specific to your plugin so that it can be customized.

Resource File

You can see that in initGui() we’ve used an icon from the resource file (called resources.qrc in our case)

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

It is good to use a prefix that will not collide with other plugins or any parts of QGIS, otherwise you might get resources you did not want. Now you just need to generate a Python file that will contain the resources. It’s done with pyrcc5 command:

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

And that’s all—nothing complicated ;)

If you’ve done everything correctly you should be able to find and load your plugin in the plugin manager and see a message in console when toolbar icon or appropriate menu item is selected.

When working on a real plugin it’s wise to write the plugin in another (working) directory and create a makefile which will generate UI + resource files and install the plugin into your QGIS installation.

16.1.3 Documentation

The documentation for the plugin can be written as HTML help files. The qgis.utils module provides a function, showPluginHelp() which will open the help file browser, in the same way as other QGIS help.

The showPluginHelp() function looks for help files in the same directory as the calling module. It will look for, in turn, index-ll_cc.html, index-ll.html, index-en.html, index-en_us.html and index.html, displaying whichever it finds first. Here ll_cc is the QGIS locale. This allows multiple translations of the documentation to be included with the plugin.

The showPluginHelp() function can also take parameters packageName, which identifies a specific plugin for which the help will be displayed, filename, which can replace 《index》 in the names of files being searched, and section, which is the name of an html anchor tag in the document on which the browser will be positioned.
16.1.4 Translation

With a few steps you can set up the environment for the plugin localization so that depending on the locale settings of your computer the plugin will be loaded in different languages.

Software requirements

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
```

Files and directory

When you create the plugin you will find the `i18n` folder within the main plugin directory.

All the translation files have to be within this directory.

.pro file

First you should create a `.pro` file, that is a `project` file that can be managed by Qt Linguist.

In this `.pro` file you have to specify all the files and forms you want to translate. This file is used to set up the localization files and variables. A possible project file, matching the structure of our example plugin:

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

Your plugin might follow a more complex structure, and it might be distributed across several files. If this is the case, keep in mind that pylupdate5, the program we use to read the `.pro` file and update the translatable string, does not expand wild card characters, so you need to place every file explicitly in the `.pro` file. Your project file might then look like something like this:

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

Furthermore, the `your_plugin.py` file is the file that `calls` all the menu and sub-menus of your plugin in the QGIS toolbar and you want to translate them all.

Finally with the `TRANSLATIONS` variable you can specify the translation languages you want.

경고: Be sure to name the `ts` file like `your_plugin_ + language + .ts` otherwise the language loading will fail! Use the 2 letter shortcut for the language (`it` for Italian, `de` for German, etc...).
Once you have created the .pro you are ready to generate the .ts file(s) for the language(s) of your plugin.

Open a terminal, go to your_plugin/i18n directory and type:

```
pylupdate5 your_plugin.pro
```

you should see the your_plugin_language.ts file(s).

Open the .ts file with Qt Linguist and start to translate.

### .qm file

When you finish to translate your plugin (if some strings are not completed the source language for those strings will be used) you have to create the .qm file (the compiled .ts file that will be used by QGIS).

Just open a terminal cd in your_plugin/i18n directory and type:

```
lrelease your_plugin.ts
```

now, in the i18n directory you will see the your_plugin.qm file(s).

### Translate using Makefile

Alternatively you can use the makefile to extract messages from python code and Qt dialogs, if you created your plugin with Plugin Builder. At the beginning of the Makefile there is a LOCALES variable:

```
LOCALES = en
```

Add the abbreviation of the language to this variable, for example for Hungarian language:

```
LOCALES = en hu
```

Now you can generate or update the hu.ts file (and the en.ts too) from the sources by:

```
make transup
```

After this, you have updated .ts file for all languages set in the LOCALES variable. Use Qt Linguist to translate the program messages. Finishing the translation the .qm files can be created by the transcompile:

```
make transcompile
```

You have to distribute .ts files with your plugin.
Load the plugin

In order to see the translation of your plugin, open QGIS, change the language (Settings → Options → General) and restart QGIS.

You should see your plugin in the correct language.

**경고**: If you change something in your plugin (new UIs, new menu, etc..) you have to generate again the update version of both .ts and .qm file, so run again the command of above.

### 16.1.5 Tips and Tricks

#### Plugin Reloader

During development of your plugin you will frequently need to reload it in QGIS for testing. This is very easy using the Plugin Reloader plugin. You can find it with the Plugin Manager.

#### Accessing Plugins

You can access all the classes of installed plugins from within QGIS using python, which can be handy for debugging purposes.

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

#### Log Messages

Plugins have their own tab within the log_message_panel.

#### Share your plugin

QGIS is hosting hundreds of plugins in the plugin repository. Consider sharing yours! It will extend the possibilities of QGIS and people will be able to learn from your code. All hosted plugins can be found and installed from within QGIS with the Plugin Manager.

Information and requirements are here: plugins.qgis.org.

### 16.2 코드 조각

- 단축키로 함수 메소드를 호출하는 방법
- 레이어를 켜고 끄는 방법
- 선택한 피처의 속성 테이블에 접근하는 방법
- 옵션 대화창에 있는 플러그인 인터페이스

이 장에서는 플러그인 개발을 쉽게 할 수 있도록 도와주는 코드 조각 (snippet) 에 대해 설명합니다.
16.2.1 단축키로 함수 메소드를 호출하는 방법

플러그인 내부의 initGui() 함수에 다음 코드를 추가하십시오.

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

unload() 함수에 다음 코드를 추가하십시오.

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

CTRL+I 단축키를 누르면 해당 메소드를 호출합니다.

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

16.2.2 레이어를 켜고 끄는 방법

범례에 있는 레이어에 접근할 수 있는 API가 있습니다. 다음은 활성화 레이어의 가시성을 켜고 꺼는 예시입니다.

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

16.2.3 선택한 피처의 속성 테이블에 접근하는 방법

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

# The method requires one parameter (the new value for the second
# field of the selected feature(s)) and can be called by
change_value(50)
16.2.4 옵션 대화창에 있는 플러그인용 인터페이스

Settings и Options 에 사용자 지정 플러그인 옵션을 추가할 수 있습니다. 사용자 플러그인의 옵션을 위해 특정 주 메뉴 항목을 추가하는 것보다는 이 편이 좋을 수 있습니다. 모든 QGIS 응용 프로그램 설정과 플러그인 설정을 단일 위치에 모아두기 때문에 사용자가 설정을 찾고 탐색하기에 더 편하기 때문입니다.

다음 조각은 플러그인의 설정에 새로운 빈 탭만 추가할 것입니다. 사용자가 이 탭에 사용자 플러그인에 특화된 모든 옵션 및 설정을 재워 넣을 수 있습니다. 다음 클래스들은 서로 다른 파일로 분할할 수 있습니다. 이 예제에서 우리는 주 mainPlugin.py 파일 안에 클래스 2 개를 추가할 것입니다.

```python
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)
```

마지막으로 가져온 것들을 추가하고 __init__ 함수를 수정합니다:

```python
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: QgisInterface
        """
        # 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)
```

팁: 벡터 레이어 속성 대화창에 사용자 지정 탭 추가

비슷한 논리를 적용해서 QgsMapLayerConfigWidgetFactory 및 QgsMapLayerConfigWidget 클래스를 사용하면 레이어 속성 대화창에 플러그인 사용자 지정 옵션을 추가할 수 있습니다.

16.3 플러그인 레이어 사용

사용자 플러그인이 맵 레이어를 렌더링하기 위해 자신만의 메소드를 이용하는 경우, 이를 구현하는 데 QgsPluginLayer 클래스에 기반한 사용자만의 레이어 유형의 레이어 없음을 작성하는 것이 가장 좋을 수도 있습니다.

16.3.1 QgsPluginLayer 상속 클래스 만들기

다음은 최소한의 QgsPluginLayer 클래스를 구현하는 예시입니다. 워터마크 예제 플러그인의 원본 코드를 기반으로 하고 있습니다.

이 구현의 일부로써, 사용자 지정 렌더링 작업자가 캔버스에 실제 그리지는 내용을 정의합니다.

```python
class WatermarkLayerRenderer(QgsMapLayerRenderer):
    def __init__(self, layerId, rendererContext):
        super().__init__(layerId, rendererContext)

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

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, 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, self, node, doc, context):
        pass
```

16.3. 플러그인 레이어 사용 127
다른 모든 맵 레이어와 마찬가지로, 프로젝트 및 캔버스에 플러그인 레이어를 추가할 수 있습니다.

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

이런 레이어를 가지고 있는 프로젝트를 불러오는 경우, 팩토리 클래스가 필요합니다:

```python
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)
```

## 16.4 IDE settings for writing and debugging plugins

- Useful plugins for writing Python plugins
- A note on configuring your IDE on Linux and Windows
- Debugging using Pyscripter IDE (Windows)
- Debugging using Eclipse and PyDev
  - 설치
  - Setting up Eclipse
  - Configuring the debugger
  - Making eclipse understand the API
- Debugging with PyCharm on Ubuntu with a compiled QGIS
- Debugging using PDB

Although each programmer has his preferred IDE/Text editor, here are some recommendations for setting up popular IDE’s for writing and debugging QGIS Python plugins.
16.4.1 Useful plugins for writing Python plugins

Some plugins are convenient when writing Python plugins. From Plugins → Manage and Install plugins..., install:

- **Plugin reloader**: This will let you reload a plugin and pull new changes without restarting QGIS.
- **First Aid**: This will add a Python console and local debugger to inspect variables when an exception is raised from a plugin.

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

16.4.2 A note on configuring your IDE on Linux and Windows

**On Linux**, all that usually needs to be done is to add the QGIS library locations to the user’s PYTHONPATH environment variable. Under most distributions, this can be done by editing ~/.bashrc or ~/.bash-profile with the following line (tested on OpenSUSE Tumbleweed):

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

Save the file and implement the environment settings by using the following shell command:

```
source ~/.bashrc
```

**On Windows**, you need to make sure that you have the same environment settings and use the same libraries and interpreter as QGIS. The fastest way to do this is to modify the startup batch file of QGIS.

If you used the OSGeo4W Installer, you can find this under the bin folder of your OSGeo4W install. Look for something like C:\OSGeo4W\bin\qgis-unstable.bat.

16.4.3 Debugging using Pyscripter IDE (Windows)

For using Pyscripter IDE, here’s what you have to do:

1. Make a copy of qgis-unstable.bat and rename it pyscripter.bat.
2. Open it in an editor. And remove the last line, the one that starts QGIS.
3. Add a line that points to your Pyscripter executable and add the command line argument that sets the version of Python to be used
4. Also add the argument that points to the folder where Pyscripter can find the Python dll used by QGIS, you can find this under the bin folder of your OSGeo4W install
5. Now when you double click this batch file it will start Pyscripter, with the correct path.
More popular than Pyscripter, Eclipse is a common choice among developers. In the following section, we will be explaining how to configure it for developing and testing plugins.

16.4.4 Debugging using Eclipse and PyDev

설치

To use Eclipse, make sure you have installed the following

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

- You may also want to install Remote Debug, a QGIS plugin. At the moment it’s still experimental so enable Experimental plugins under Plugins → Manage and Install plugins → Options beforehand.

To prepare your environment for using Eclipse in Windows, you should also create a batch file and use it to start Eclipse:

1. Locate the folder where qgis_core.dll resides in. Normally this is C:\OSGeo4W\apps\qgis\bin, but if you compiled your own QGIS application this is in your build folder in output/bin/RelWithDebInfo
2. Locate your eclipse.exe executable.
3. Create the following script and use this to start eclipse when developing QGIS plugins.

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

Setting up Eclipse

1. In Eclipse, create a new project. You can select General Project and link your real sources later on, so it does not really matter where you place this project.
2. Right-click your new project and choose New → Folder.
3. Click Advanced and choose Link to alternate location (Linked Folder). In case you already have sources you want to debug, choose these. In case you don’t, create a folder as it was already explained.

Now in the view Project Explorer, your source tree pops up and you can start working with the code. You already have syntax highlighting and all the other powerful IDE tools available.

Configuring the debugger

To get the debugger working:

1. Switch to the Debug perspective in Eclipse (Window → Open Perspective → Other → Debug).
2. start the PyDev debug server by choosing PyDev → Start Debug Server.
3. Eclipse is now waiting for a connection from QGIS to its debug server and when QGIS connects to the debug server it will allow it to control the python scripts. That’s exactly what we installed the Remote Debug plugin for. So start QGIS in case you did not already and click the bug symbol.
Select a wizard

Create a new project resource

Wizards:

<table>
<thead>
<tr>
<th>type filter text</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP Project</td>
</tr>
<tr>
<td>Rails Project</td>
</tr>
<tr>
<td>Ruby Project</td>
</tr>
<tr>
<td>Web Project</td>
</tr>
<tr>
<td>General</td>
</tr>
<tr>
<td>Project</td>
</tr>
<tr>
<td>C/C++</td>
</tr>
<tr>
<td>CVS</td>
</tr>
<tr>
<td>Java</td>
</tr>
<tr>
<td>PyDev</td>
</tr>
<tr>
<td>Ruby</td>
</tr>
<tr>
<td>Web</td>
</tr>
<tr>
<td>Other</td>
</tr>
</tbody>
</table>

그림 16.1: Eclipse project

16.4. IDE settings for writing and debugging plugins 131
Now you can set a breakpoint and as soon as the code hits it, execution will stop and you can inspect the current state of your plugin. (The breakpoint is the green dot in the image below, set one by double clicking in the white space left to the line you want the breakpoint to be set).

A very interesting thing you can make use of now is the debug console. Make sure that the execution is currently stopped at a break point, before you proceed.

1. Open the Console view (Window → Show view). It will show the Debug Server console which is not very interesting. But there is a button Open Console which lets you change to a more interesting PyDev Debug Console.

2. Click the arrow next to the Open Console button and choose PyDev Console. A window opens up to ask you which console you want to start.

3. Choose PyDev Debug Console. In case its greyed out and tells you to Start the debugger and select the valid frame, make sure that you’ve got the remote debugger attached and are currently on a breakpoint.

You have now an interactive console which lets you test any commands from within the current context. You can manipulate variables or make API calls or whatever you like.

Tip: A little bit annoying is, that every time you enter a command, the console switches back to the Debug Server. To stop this behavior, you can click the Pin Console button when on the Debug Server page and it should remember this decision at least for the current debug session.
Making eclipse understand the API

A very handy feature is to have Eclipse actually know about the QGIS API. This enables it to check your code for typos. But not only this, it also enables Eclipse to help you with autocompletion from the imports to API calls.

To do this, Eclipse parses the QGIS library files and gets all the information out there. The only thing you have to do is to tell Eclipse where to find the libraries.

1. Click **Window ➤ Preferences ➤ PyDev ➤ Interpreter ➤ Python**.

   You will see your configured python interpreter in the upper part of the window (at the moment python2.7 for QGIS) and some tabs in the lower part. The interesting tabs for us are **Libraries** and **Forced Builtins**.

   ![PyDev Debug Console](image)

   **그림 16.4: PyDev Debug Console**

2. First open the Libraries tab.
3. Add a New Folder and choose the python folder of your QGIS installation. If you do not know where this folder is (it’s not the plugins folder):
   1. Open QGIS
   2. Start a python console
   3. Enter `qgis`
   4. and press Enter. It will show you which QGIS module it uses and its path.
   5. Strip the trailing `/qgis/__init__.pyc` from this path and you’ve got the path you are looking for.
4. You should also add your plugins folder here (it is in `python/plugins` under the user profile folder).

5. Next jump to the Forced Builtins tab, click on New… and enter `qgis`. This will make Eclipse parse the QGIS API. You probably also want Eclipse to know about the PyQt API. Therefore also add PyQt as forced builtin. That should probably already be present in your libraries tab.

6. Click OK and you’re done.

참고: Every time the QGIS API changes (e.g. if you’re compiling QGIS master and the SIP file changed), you should go back to this page and simply click Apply. This will let Eclipse parse all the libraries again.

### 16.4.5 Debugging with PyCharm on Ubuntu with a compiled QGIS

PyCharm is an IDE for Python developed by JetBrains. There is a free version called Community Edition and a paid one called Professional. You can download PyCharm on the website: [https://www.jetbrains.com/pycharm/download](https://www.jetbrains.com/pycharm/download)

We are assuming that you have compiled QGIS on Ubuntu with the given build directory `~/dev/qgis/build/master`. It’s not compulsory to have a self compiled QGIS, but only this has been tested. Paths must be adapted.

1. In PyCharm, in your Project Properties, Project Interpreter, we are going to create a Python Virtual environment called QGIS.

2. Click the small gear and then Add.

3. Select Virtualenv environment.

4. Select a generic location for all your Python projects such as `~/dev/qgis/venv` because we will use this Python interpreter for all our plugins.

5. Choose a Python 3 base interpreter available on your system and check the next two options Inherit global site-packages and Make available to all projects.

![Add Python Interpreter](image)

1. Click OK, come back on the small gear and click Show all.

2. In the new window, select your new interpreter QGIS and click the last icon in the vertical menu Show paths for the selected interpreter.

3. Finally, add the following absolute path to the list `~/dev/qgis/build/master/output/python`. 

---

[134] Chapter 16. 파이썬 플러그인 개발
16.4. IDE settings for writing and debugging plugins
1. Restart PyCharm and you can start using this new Python virtual environment for all your plugins.

PyCharm will be aware of the QGIS API and also of the PyQt API if you use Qt provided by QGIS like `from qgis. PyQt.QtCore import QDir`. The autocompletion should work and PyCharm can inspect your code.

In the professional version of PyCharm, remote debugging is working well. For the Community edition, remote debugging is not available. You can only have access to a local debugger, meaning that the code must run inside PyCharm (as script or unittest), not in QGIS itself. For Python code running in QGIS, you might use the *First Aid* plugin mentioned above.

### 16.4.6 Debugging using PDB

If you do not use an IDE such as Eclipse or PyCharm, you can debug using PDB, following these steps.

1. First add this code in the spot where you would like to debug

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

2. Then run QGIS from the command line.

   **On Linux do:**
   
   ```bash
   $ ./Qgis
   ```

   **On macOS do:**
   
   ```bash
   $ /Applications/Qgis.app/Contents/MacOS/Qgis
   ```

3. And when the application hits your breakpoint you can type in the console!

**TODO:** Add testing information

### 16.5 Releasing your plugin

- **Metadata and names**
- **Code and help**
- **Official Python plugin repository**
  - Permissions
  - Trust management
  - Validation
  - Plugin structure

Once your plugin is ready and you think the plugin could be helpful for some people, do not hesitate to upload it to *Official Python plugin repository*. On that page you can also find packaging guidelines about how to prepare the plugin to work...
well with the plugin installer. Or in case you would like to set up your own plugin repository, create a simple XML file that will list the plugins and their metadata. Please take special care to the following suggestions:

16.5.1 Metadata and names

- avoid using a name too similar to existing plugins
- if your plugin has a similar functionality to an existing plugin, please explain the differences in the About field, so the user will know which one to use without the need to install and test it
- avoid repeating ‘plugin’ in the name of the plugin itself
- use the description field in metadata for a 1 line description, the About field for more detailed instructions
- include a code repository, a bug tracker, and a home page; this will greatly enhance the possibility of collaboration, and can be done very easily with one of the available web infrastructures (GitHub, GitLab, Bitbucket, etc.)
- choose tags with care: avoid the uninformative ones (e.g. vector) and prefer the ones already used by others (see the plugin website)
- add a proper icon, do not leave the default one; see QGIS interface for a suggestion of the style to be used

16.5.2 Code and help

- do not include generated file (ui_*.py, resources_rc.py, generated help files… ) and useless stuff (e.g. .gitignore) in repository
- add the plugin to the appropriate menu (Vector, Raster, Web, Database)
- when appropriate (plugins performing analyses), consider adding the plugin as a subplugin of Processing framework: this will allow users to run it in batch, to integrate it in more complex workflows, and will free you from the burden of designing an interface
- include at least minimal documentation and, if useful for testing and understanding, sample data.

16.5.3 Official Python plugin repository

You can find the official Python plugin repository at https://plugins.qgis.org/. In order to use the official repository you must obtain an OSGEO ID from the OSGEO web portal. Once you have uploaded your plugin it will be approved by a staff member and you will be notified.

TODO: Insert a link to the governance document

Permissions

These rules have been implemented in the official plugin repository:

- every registered user can add a new plugin
- staff users can approve or disapprove all plugin versions
- users which have the special permission plugins.can_approve get the versions they upload automatically approved
- users which have the special permission plugins.can_approve can approve versions uploaded by others as long as they are in the list of the plugin owners

16.5. Releasing your plugin
• a particular plugin can be deleted and edited only by staff users and plugin owners
• if a user without plugins.can_approve permission uploads a new version, the plugin version is automatically unapproved.

Trust management

Staff members can grant trust to selected plugin creators setting plugins.can_approve permission through the front-end application.

The plugin details view offers direct links to grant trust to the plugin creator or the plugin owners.

Validation

Plugin’s metadata are automatically imported and validated from the compressed package when the plugin is uploaded. Here are some validation rules that you should aware of when you want to upload a plugin on the official repository:

1. the name of the main folder containing your plugin must contain only ASCII characters (A-Z and a-z), digits and the characters underscore (_) and minus (-), also it cannot start with a digit
2. metadata.txt is required
3. all required metadata listed in metadata table must be present
4. the version metadata field must be unique

Plugin structure

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 its own directory inside the .zip file to not interfere with other plugins. Mandatory files are: metadata.txt and __init__.py. But it would be nice to have a README and of course an icon to represent the plugin (resources.qrc). Following is an example of how a plugin.zip should look like.

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

파이썬 프로그래밍 언어로 플러그인을 생성할 수 있습니다. C++로 작성된 전통적인 플러그인에 비해 더 쉽게 플러그인을 작성하고, 이해하고, 유지하고 배포할 수 있습니다. 이는 파이썬 언어의 동적인 특성 덕분입니다.

파이썬 플러그인 목록은 QGIS 플러그인 관리자에서 C++ 플러그인과 함께 볼 수 있습니다. */(UserProfile)/python/plugins 및 다음 경로에서 플러그인을 검색합니다:
• 유닉스/맥: (qgis_prefix)/share/qgis/python/plugins
• 윈도우: (qgis_prefix)/python/plugins

~ 및 (UserProfile) 이 어떻게 정의되는지 알고 싶다면 core_and_external_plugins 을 참조하세요.

참고: 기존 디렉터이 경로에 QGIS_PLUGINPATH 를 설정하면, 플러그인 검색 경로 목록에 이 경로를 추가할 수 있습니다.
CHAPTER 17

Writing a Processing plugin

- Creating from scratch
- Updating a plugin

Depending on the kind of plugin that you are going to develop, it might be a better option to add its functionality as a Processing algorithm (or a set of them). That would provide a better integration within QGIS, additional functionality (since it can be run in the components of Processing, such as the modeler or the batch processing interface), and a quicker development time (since Processing will take of a large part of the work).

To distribute those algorithms, you should create a new plugin that adds them to the Processing Toolbox. The plugin should contain an algorithm provider, which has to be registered when the plugin is instantiated.

17.1 Creating from scratch

To create a plugin from scratch which contains an algorithm provider, you can follow these steps using the Plugin Builder:

1. Install the Plugin Builder plugin
2. Create a new plugin using the Plugin Builder. When the Plugin Builder asks you for the template to use, select 《Processing provider》.
3. The created plugin contains a provider with a single algorithm. Both the provider file and the algorithm file are fully commented and contain information about how to modify the provider and add additional algorithms. Refer to them for more information.
17.2 Updating a plugin

If you want to add your existing plugin to Processing, you need to add some code.

1. In your `metadata.txt` file, you need to add a variable:

   ```
   hasProcessingProvider=yes
   ```

2. In the Python file where your plugin is setup with the `initGui` method, you need to adapt some lines like this:

   ```
   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. You can create a folder `processing_provider` with three files in it:

   • `__init__.py` with nothing in it. This is necessary to make a valid Python package.
   • `provider.py` which will create the Processing provider and expose your algorithms.

   ```
   from qgis.core import QgsProcessingProvider

   from processing_provider.example_processing_algorithm import...
   # ExampleProcessingAlgorithm

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

       def id(self, *args, **kwars):
           """The ID of your plugin, used for identifying the provider.
           This string should be a unique, short, character only string, eg "qgis" or "gdal". This string should not be localised.
           ""
           return 'yourplugin'

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

   (다음을 계속)
This string should be as short as possible (e.g. "Lastools", not "Lastools version 1.0.1 64-bit") and localised.

```python
return self.tr('Your plugin')
```

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

- `example_processing_algorithm.py` which contains the example algorithm file. Copy/paste the content of the script template file and update it according to your needs.

4. Now you can reload your plugin in QGIS and you should see your example script in the Processing toolbox and modeler.

The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.core import 
QgsVectorLayer,
QgsPointXY,
```

17.2. Updating a plugin
네트워크 분석 라이브러리

• 일반 정보
• 그래프 만들기
• 그래프 분석
  - 최단 경로 탐색
  - 도달 가능 범위

The network analysis library can be used to:
  • create mathematical graph from geographical data (polyline vector layers)
  • implement basic methods from graph theory (currently only Dijkstra’s algorithm)

네트워크 분석 라이브러리는 코어 플러그인 RoadGraph 에서 기본 함수들을 가져와 생성됐습니다. 이제 플러그인에서나 파이썬 콘솔에서 직접 이 라이브러리의 메소드를 사용할 수 있습니다.

18.1 일반 정보

이 라이브러리의 전형적인 용도를 간단히 설명하면 다음과 같습니다.
  1. 공간 데이터 (일반적으로 폴리라인 벡터 레이어) 로부터 그래프 생성
  2. 그래프 분석 실행
  3. 분석 결과 이용 (예를 들어 분석 결과의 시각화 등)
18.2 그래프 만들기

가장 먼저 해결해야 할 일은 입력 데이터를 준비하는 것인데, 벡터 레이어를 그래프 (수학적으로 간략화된 연결관계) 로 변환하는 것입니다. 이후의 모든 작업은 레이어가 아니라 그래프를 사용합니다.

먼저 플리라인 벡터 레이어라도 소스로 사용할 수 있습니다. 플리라인의 노드 (node) 는 그래프의 버텍스 (vertex) 가 되고, 플리라인의 선분 (segment) 은 그래프의 엣지 (edge) 가 됩니다. 노드 몇 개가 동일한 좌표에 있을 경우 그 노드들은 동일한 그래프 버텍스가 됩니다. 따라서 공통 노드를 가진 2개의 선분은 서로 연결됩니다.

또, 그래프 생성 중에 입력 벡터 레이어에 추가적인 포인트를 몇 개라도 (고정) 시킬 수 있습니다. 각 추가 포인트에 대응하는, 가장 가까운 그래프 버텍스 또는 가장 가까운 그래프 엣지를 찾을 것입니다. 향후의 경우 엣지가 나뉘어 새 버텍스가 추가됩니다.

벡터 레이어의 속성과 엣지 길이를 그래프 엣지의 속성으로 쓸 수 있습니다.

Converting from a vector layer to the graph is done using the Builder programming pattern. A graph is constructed using a so-called Director. There is only one Director for now: QgsVectorLayerDirector. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create the graph. Currently, as in the case with the director, only one builder exists: QgsGraphBuilder, that creates QgsGraph objects. You may want to implement your own builders that will build a graph compatible with such libraries as BGL or NetworkX.

To calculate edge properties the programming pattern strategy is used. For now only QgsNetworkDistanceStrategy strategy (that takes into account the length of the route) and QgsNetworkSpeedStrategy (that also considers the speed) are available. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses a strategy that computes travel time using edge length and speed value from attributes.

이제 실제로 해 볼습니다.

First of all, to use this library we should import the analysis module

```python
from qgis.analysis import *
```

d음은 director를 생성하는 몇 가지 방법의 예시입니다.

```python
# 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)
```

To construct a director, we should pass a vector layer that will be used as the source for the graph structure and information about allowed movement on each road segment (one-way or bidirectional movement, direct or reverse direction). The call looks like this

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

g각의 파라미터가 의미하는 바는 다음과 같습니다.
• **vectorLayer** — vector layer used to build the graph

• **directionFieldId** — 도로 방향에 관한 정보가 저장된 속성 테이블 필드의 인덱스. 값이 -1 일 경우 방향 정보를 전혀 사용하지 않습니다. 정수형입니다.

• **directDirectionValue** — 순방향 (첫 번째 라인 포인트에서 마지막 라인 포인트로 이동) 인 도로의 필드값. 문자열입니다.

• **reverseDirectionValue** — 역방향 (마지막 라인 포인트에서 첫 번째 포인트로 이동) 인 도로의 필드값. 문자열입니다.

• **bothDirectionValue** — 양방향 (첫 번째 포인트에서 마지막으로도 마지막에서 첫 번째로도 이동 가능) 인 도로의 필드값. 문자열입니다.

• **defaultDirection** — default road direction. This value will be used for those roads where field directionFieldId is not set or has some value different from any of the three values specified above. Possible values are:
  - QgsVectorLayerDirector.DirectionForward — One-way direct
  - QgsVectorLayerDirector.DirectionBackward — One-way reverse
  - QgsVectorLayerDirector.DirectionBoth — Two-way

그 다음 **edge 속성을 계산하기 위해 strategy 를 생성해야 합니다.**

```python
# 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)
```

그리고 **director** 에게 이 strategy 에 대해 알려줍니다.

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

Now we can use the builder, which will create the graph. The **QgsGraphBuilder** class constructor takes several arguments:

• **crs** — coordinate reference system to use. Mandatory argument.

• **otfEnabled** — use 《on the fly》 reprojection or no. By default True (use OTF).

• **topologyTolerance** — topological tolerance. Default value is 0.

• **ellipsoidID** — ellipsoid to use. By default 《WGS84》.

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

또 분석 작업에 사용할 포인트를 몇 개 다음과 같이 정의합나다.

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

이제 모든 준비가 끝났으므로 그래프를 만들고 이 포인트들을 그래프에 《결속 (tie)》 시킬 수 있습니다.

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

18.2. 그래프 만들기 147
18.3 그래프 분석

네트워크 분석은 다음 두 가지 질문에 대한 답을 찾는 데 사용됩니다. 어떤 vertex 들이 연결되어 있는가? 그리고 어떻게 최단 경로를 찾을 것인가? 네트워크 분석 라이브러리는 이 문제를 해결하기 위해 데이크스트라 알고리즘 (Dijkstra’s algorithm) 을 제공합니다.

데이크스트라 알고리즘은 그래프의 한 vertex 에서 다른 모든 vertex 로 가는 최단 경로와 최적화 파라미터의 값을 찾습니다. 그 결과는 최단 경로 트리로 나타낼 수 있습니다.

The shortest path tree is a directed weighted graph (or more precisely a tree) with the following properties:

• 들어오는 edge 가 없는 vertex 는 단 하나, 트리의 루트뿐입니다.
• 다른 모든 vertex 는 들어오는 edge 를 막 하나 가지고 있습니다.
• vertex A 에서 vertex B 에 도달할 수 있다면, A 에서 B 로의 경로는 사용할 수 있는 단 하나의 경로이며 이 그래프에서 최적 (최단) 경로입니다.

To get the shortest path tree use the methods shortestTree() and dijkstra() of the QgsGraphAnalyzer class. It is recommended to use the dijkstra() method because it works faster and uses memory more efficiently.

The shortestTree() method is useful when you want to walk around the shortest path tree. It always creates a new graph object (QgsGraph) and accepts three variables:

- source — input graph
- startVertexIdx — index of the point on the tree (the root of the tree)
- criterionNum — number of edge property to use (started from 0).

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

The dijkstra() method has the same arguments, but returns two arrays. In the first array element \( n \) contains index of the incoming edge or -1 if there are no incoming edges. In the second array element \( n \) contains the distance from the root of the tree to vertex \( n \) or DOUBLE_MAX if vertex \( n \) is unreachable from the root.

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

Here is some very simple code to display the shortest path tree using the graph created with the shortestTree() method (select linestring layer in Layers panel and replace coordinates with your own).

경고: Use this code only as an example, it creates a lot of QgsRubberBand objects and may be slow on large datasets.
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines', 'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.←DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())
pStart = QgsPointXY(1179661.925139, 5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()
idStart = graph.findVertex(pStart)
tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)
i = 0
while (i < tree.edgeCount()):
    rb = QgsRubberBand(iface.mapCanvas())
    rb.setColor(Qt.red)
    rb.addPoint(tree.vertex(tree.edge(i).fromVertex()).point())
    rb.addPoint(tree.vertex(tree.edge(i).toVertex()).point())
    i = i + 1

Same thing but using the dijkstra() method

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines', 'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.←DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())
pStart = QgsPointXY(1179661.925139, 5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

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

(다음 페이지에 계속)
for edgeId in tree:
    if edgeId == -1:
        continue
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.red)
rb.addPoint(graph.vertex(graph.edge(edgeId).fromVertex()).point())
rb.addPoint(graph.vertex(graph.edge(edgeId).toVertex()).point())

18.3.1 최단 경로 탐색

To find the optimal path between two points the following approach is used. Both points (start A and end B) are tied to the graph when it is built. Then using the shortestTree() or dijkstra() method we build the shortest path tree with root in the start point A. In the same tree we also find the end point B and start to walk through the tree from point B to point A. The whole algorithm can be written as:

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

이 시점에서 이 경로를 지나가는 동안 거치게 될 vertex 의 역순 목록의 형태로 경로를 얻게 됩니다. (vertex 들이 종료점에서 시작점의 순서로 역순으로 나열됩니다.)

Here is the sample code for QGIS Python Console (you may need to load and select a linestring layer in TOC and replace coordinates in the code with yours) that uses the shortestTree() method

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)
startPoint = QgsPointXY(1179661.925139, 5419188.074362)
endPoint = QgsPointXY(1180942.970617, 5420040.097560)
tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
tStart, tStop = tiedPoints
tStart = 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])
    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)

And here is the same sample but using the dijkstra() method

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!')
18.3.2 도달 가능 범위

vertex A의 도달 가능 범위(area of availability)란 vertex A에서 접근할 수 있고, vertex A에서 이 vertex들까지의 경로비용이 지정된 값을 초과하지 않는, 그래프 vertex들의 부분집합을 말합니다.

다음 질문을 통해 이를 더 명확히 알 수 있습니다. 《소방서가 있다. 소방차가 5분/10분/15분 안에 도착할 수 있는 도시의 구역은 어디인가?》이 질문에 대한 답이 바로 소방서의 도달 가능 범위입니다.

To find the areas of availability we can use the dijkstra() method of the QgsGraphAnalyzer class. It is enough to compare the elements of the cost array with a predefined value. If cost[i] is less than or equal to a predefined value, then vertex i is inside the area of availability, otherwise it is outside.

도달 가능 범위의 경계를 구하는 일은 좀 더 어려운 문제입니다. 한단 경계는 도달 가능한 vertex들의 집합이고, 상단 경계는 도달 불가능한 vertex들의 집합입니다. 사실 단순합니다. 도달 가능 범위의 경계는 edge의 원본 vertex가 접근 가능한 vertex이고, edge의 대상 vertex가 접근 불가능한 vertex인 최단경로 트리의 edge들에 가반한 경계선입니다.

다음은 그 예시입니다.

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(), True)
rb.setColor(Qt.green)
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
r.addPoint(QgsPointXY(pStart.x() + delta, pStart.y()) - delta))
r.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))

(다음 페이지에 계속)
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

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

18.3. 그래프 분석
CHAPTER 19

QGIS Server and Python

19.1 소개

To learn more about QGIS Server, read the QGIS-Server-manual.

QGIS Server is three different things:

1. QGIS Server library: a library that provides an API for creating OGC web services
2. QGIS Server FCGI: a FCGI binary application `qgis_maserv.fcgi` that together with a web server implements a set of OGC services (WMS, WFS, WCS etc.) and OGC APIs (WFS3/OAPIF)
3. QGIS Development Server: a development server binary application `qgis_mapserver` that implements a set of OGC services (WMS, WFS, WCS etc.) and OGC APIs (WFS3/OAPIF)

This chapter of the cookbook focuses on the first topic and by explaining the usage of QGIS Server API it shows how it is possible to use Python to extend, enhance or customize the server behavior or how to use the QGIS Server API to embed QGIS server into another application.

There are a few different ways you can alter the behavior of QGIS Server or extend its capabilities to offer new custom services or APIs, these are the main scenarios you may face:

- **EMBEDDING** → Use QGIS Server API from another Python application
- **STANDALONE** → Run QGIS Server as a standalone WSGI/HTTP service
- **FILTERS** → Enhance/Customize QGIS Server with filter plugins
- **SERVICES** → Add a new **SERVICE**
- **OGC APIs** → Add a new **OGC API**

Embedding and standalone applications require using the QGIS Server Python API directly from another Python script or application. The remaining options are better suited for when you want to add custom features to a standard QGIS Server binary application (FCGI or development server): in this case you’ll need to write a Python plugin for the server application and register your custom filters, services or APIs.
19.2 Server API basics

The fundamental classes involved in a typical QGIS Server application are:

- QgsServer: the server instance (typically a single instance for the whole application life)
- QgsServerRequest: the request object (typically recreated on each request)
- QgsServer.handleRequest(request, response): processes the request and populates the response

The QGIS Server FCGI or development server workflow can be summarized as follows:

```python
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
```

Inside the QgsServer.handleRequest(request, response) method the filter plugins callbacks are called and QgsServerRequest and QgsServerResponse are made available to the plugins through the QgsServer-Interface class.

경고: QGIS server classes are not thread safe, you should always use a multiprocessing model or containers when building scalable applications based on QGIS Server API.

19.3 Standalone or embedding

For standalone server applications or embedding, you will need to use the above mentioned server classes directly, wrapping them up into a web server implementation that manages all the HTTP protocol interactions with the client.

A minimal example of the QGIS Server API usage (without the HTTP part) follows:

```python
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
```

(다음 페이지에 계속)
Here is a complete standalone application example developed for the continuous integrations testing on QGIS source code repository, it showcases a wide set of different plugin filters and authentication schemes (not mean for production because they were developed for testing purposes only but still interesting for learning): https://github.com/qgis/QGIS/blob/release-3_16/tests/src/python/qgis_wrapped_server.py

19.4 Server plugins

Server python plugins are loaded once when the QGIS Server application starts and can be used to register filters, services or APIs.

The structure of a server plugin is very similar to their desktop counterpart, a QgsServerInterface object is made available to the plugins and the plugins can register one or more custom filters, services or APIs to the corresponding registry by using one of the methods exposed by the server interface.

19.4.1 Server filter plugins

Filters come in three different flavors and they can be instantiated by subclassing one of the classes below and by calling the corresponding method of QgsServerInterface:

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Base Class</th>
<th>QgsServerInterface registration</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>QgsServerFilter</td>
<td>registerFilter()</td>
</tr>
<tr>
<td>Access Control</td>
<td>QgsAccessControlFilter</td>
<td>registerAccessControl()</td>
</tr>
<tr>
<td>Cache</td>
<td>QgsServerCacheFilter</td>
<td>registerServerCache()</td>
</tr>
</tbody>
</table>

I/O filters

I/O filters can modify the server input and output (the request and the response) of the core services (WMS, WFS etc.) allowing to do any kind of manipulation of the services workflow. It is possible for example to restrict the access to selected layers, to inject an XSL stylesheet to the XML response, to add a watermark to a generated WMS image and so on.

From this point, you might find useful a quick look to the server plugins API docs.

Each filter should implement at least one of three callbacks:

- requestReady()
- responseComplete()
- sendResponse()

All filters have access to the request/response object (QgsRequestHandler) and can manipulate all its properties (input/output) and raise exceptions (while in a quite particular way as we’ll see below).

Here is the pseudo code showing how the server handles a typical request and when the filter’s callbacks are called:
for each incoming request:
    create GET/POST request handler
    pass request to an instance of QgsServerInterface
    call requestReady filters
    if there is not a response:
        if SERVICE is WMS/WFS/WCS:
            create WMS/WFS/WCS service
            call service’s executeRequest
            possibly call sendResponse for each chunk of bytes
            sent to the client by a streaming services (WFS)
        call responseComplete
        call sendResponse
    request handler sends the response to the client

The following paragraphs describe the available callbacks in details.

requestReady

This is called when the request is ready: incoming URL and data have been parsed and before entering the core services (WMS, WFS etc.) switch, this is the point where you can manipulate the input and perform actions like:

- authentication/authorization
- redirects
- add/remove certain parameters (typenames for example)
- raise exceptions

You could even substitute a core service completely by changing SERVICE parameter and hence bypassing the core service completely (not that this make much sense though).

sendResponse

This is called whenever any output is sent to FCGI stdout (and from there, to the client). This is normally done after core services have finished their process and after responseComplete hook was called, but in a few cases XML can become so huge that a streaming XML implementation is needed (WFS GetFeature is one of them). In that case, instead of a single call to sendResponse(), the method might be exceptionally called multiple times before the response is complete, and in that case (and only in that case) it is also called before responseComplete().

sendResponse() is the best place for direct manipulation of core service’s output and while responseComplete() is typically also an option, sendResponse() is the only viable option in case of streaming services.

responseComplete

This is called once when core services (if hit) finish their process and the request is ready to be sent to the client. As discussed above, this is normally called before sendResponse() except for streaming services (or other plugin filters) that might have called sendResponse() earlier.

responseComplete() is the ideal place to provide new services implementation (WPS or custom services) and to perform direct manipulation of the output coming from core services (for example to add a watermark upon a WMS image).
Raising exceptions from a plugin

Some work has still to be done on this topic: the current implementation can distinguish between handled and unhandled exceptions by setting a `QgsRequestHandler` property to an instance of `QgsMapServiceException`, this way the main C++ code can catch handled python exceptions and ignore unhandled exceptions (or better: log them).

This approach basically works but it is not very *pythonic*: a better approach would be to raise exceptions from python code and see them bubbling up into C++ loop for being handled there.

Writing a server plugin

A server plugin is a standard QGIS Python plugin as described in *파이썬 플러그인 개발*, that just provides an additional (or alternative) interface: a typical QGIS desktop plugin has access to QGIS application through the `QgisInterface` instance, a server plugin has only access to a `QgsServerInterface` when it is executed within the QGIS Server application context.

To make QGIS Server aware that a plugin has a server interface, a special metadata entry is needed (in `metadata.txt`):

```text
server=True
```

중요: Only plugins that have the `server=True` metadata set will be loaded and executed by QGIS Server.

The example plugin discussed here (with many more) is available on github at [https://github.com/elpaso/qgis3-server-vagrant/tree/master/resources/web/plugins](https://github.com/elpaso/qgis3-server-vagrant/tree/master/resources/web/plugins), a few server plugins are also published in the official QGIS plugins repository.

Plugin files

Here’s the directory structure of our example server plugin.

```plaintext
PYTHON_PLUGINS_PATH/
 HelloServer/
   __init__.py   --> *required*
   HelloServer.py --> *required*
   metadata.txt --> *required*
```

**__init__.py**

This file is required by Python’s import system. Also, QGIS Server requires that this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin’s class. This is how the example plugin `__init__.py` looks like:

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

This is where the magic happens and this is how magic looks like: (e.g. HelloServer.py)

A server plugin typically consists in one or more callbacks packed into instances of a QgsServerFilter.

Each QgsServerFilter implements one or more of the following callbacks:

- requestReady()
- responseComplete()
- sendResponse()

The following example implements a minimal filter which prints HelloServer! in case the SERVICE parameter equals to "HELLO":

```python
class HelloFilter(QgsServerFilter):
    def __init__(self, serverIface):
        super().__init__(serverIface)

    def requestReady(self):
        QgsMessageLog.logMessage("HelloFilter.requestReady")

    def sendResponse(self):
        QgsMessageLog.logMessage("HelloFilter.sendResponse")

    def responseComplete(self):
        QgsMessageLog.logMessage("HelloFilter.responseComplete")
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
            request.clear()
            request.setResponseHeader('Content-type', 'text/plain')
        request.appendBody(b'HelloServer!')
```

The filters must be registered into the serverIface as in the following example:

```python
class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(serverIface), 100)
```

The second parameter of registerFilter() sets a priority which defines the order for the callbacks with the same name (the lower priority is invoked first).

By using the three callbacks, plugins can manipulate the input and/or the output of the server in many different ways. In every moment, the plugin instance has access to the QgsRequestHandler through the QgsServerInterface. The QgsRequestHandler class has plenty of methods that can be used to alter the input parameters before entering the core processing of the server (by using requestReady()) or after the request has been processed by the core services (by using sendResponse()).

The following examples cover some common use cases:
Modifying the input

The example plugin contains a test example that changes input parameters coming from the query string, in this example a new parameter is injected into the (already parsed) `parameterMap`, this parameter is then visible by core services (WMS etc.), at the end of core services processing we check that the parameter is still there:

```python
class ParamsFilter(QgsServerFilter):
    def __init__(self, serverIface):
        super(ParamsFilter, self).__init__(serverIface)

    def requestReady(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')

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

This is an extract of what you see in the log file:

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] → HelloServer - 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: (requestReady) [0ms] QgsServerFilter plugin → default requestReady called
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] → SUCCESS - ParamsFilter.responseComplete
```

On the highlighted line the “SUCCESS” string indicates that the plugin passed the test.

The same technique can be exploited to use a custom service instead of a core one: you could for example skip a WFS SERVICE request or any other core request just by changing the SERVICE parameter to something different and the core service will be skipped. Then you can inject your custom results into the output and send them to the client (this is explained below).

❗️ If you really want to implement a custom service it is recommended to subclass `QgsService` and register your service on `registerFilter()` by calling its `registerService(service)`

---

19.4. Server plugins 161
Modifying or replacing the output

The watermark filter example shows how to replace the WMS output with a new image obtained by adding a watermark image on the top of the WMS image generated by the WMS core service:

```python
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 responseComplete(self):
        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.responseComplete: image ready \%s\n" % 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)
```

In this example the SERVICE parameter value is checked and if the incoming request is a WMS GETMAP and no exceptions have been set by a previously executed plugin or by the core service (WMS in this case), the WMS generated image is retrieved from the output buffer and the watermark image is added. The final step is to clear the output buffer and replace it with the newly generated image. Please note that in a real-world situation we should also check for the requested image type instead of supporting PNG or JPG only.
Access control filters

Access control filters gives the developer a fine-grained control over which layers, features and attributes can be accessed, the following callbacks can be implemented in an access control filter:

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

Plugin files

Here’s the directory structure of our example plugin:

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

`__init__.py`

This file is required by Python’s import system. As for all QGIS server plugins, this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server at startup. It receives a reference to an instance of `QgsServerInterface` and must return an instance of your plugin’s class. This is how the example plugin `__init__.py` looks like:

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

`AccessControl.py`

```python
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):
```

(다음 페이지에 계속)
```python
    """ 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):
        """ Register AccessControlFilter ""
        serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

This example gives a full access for everybody.
It’s the role of the plugin to know who is logged on.
On all those methods we have the layer on argument to be able to customise the restriction per layer.

**layerFilterExpression**

Used to add an Expression to limit the results, e.g.:

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

To limit on feature where the attribute role is equals to 「user」.

**layerFilterSubsetString**

Same than the previous but use the **SubsetString** (executed in the database)

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

To limit on feature where the attribute role is equals to 「user」.
layerPermissions

Limit the access to the layer.

Return an object of type `LayerPermissions()`, which has the properties:

- `canRead` to see it in the `GetCapabilities` and have read access.
- `canInsert` to be able to insert a new feature.
- `canUpdate` to be able to update a feature.
- `canDelete` to be able to delete a feature.

Example:

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

To limit everything on read only access.

authorizedLayerAttributes

Used to limit the visibility of a specific subset of attribute.

The argument attribute return the current set of visible attributes.

Example:

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

To hide the `<role>` attribute.

allowToEdit

This is used to limit the editing on a subset of features.

It is used in the WFS-Transaction protocol.

Example:

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

To be able to edit only feature that has the attribute role with the value user.
cacheKey

QGIS server maintain a cache of the capabilities then to have a cache per role you can return the role in this method. Or return None to completely disable the cache.

### 19.4.2 Custom services

In QGIS Server, core services such as WMS, WFS and WCS are implemented as subclasses of QgsService.

To implemented a new service that will be executed when the query string parameter SERVICE matches the service name, you can implemented your own QgsService and register your service on the serviceRegistry() by calling its registerService(service).

Here is an example of a custom service named CUSTOM:

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

### 19.4.3 Custom APIs

In QGIS Server, core OGC APIs such OAPIF (aka WFS3) are implemented as collections of QgsServerOgcApi-Handler subclasses that are registered to an instance of QgsServerOgcApi (or its parent class QgsServerApi).

To implemented a new API that will be executed when the url path matches a certain URL, you can implemented your own QgsServerOgcApiHandler instances, add them to an QgsServerOgcApi and register the API on the serviceRegistry() by calling its registerApi(api).

Here is an example of a custom API that will be executed when the URL contains /customapi:

```python
import json
import os

from qgis.PyQt.QtCore import QBuffer, QIODevice, QTextStream, QRegularExpression
```
from qgis.server import 
    QgsServiceRegistry,
    QgsService,
    QgsServerFilter,
    QgsServerOgcApi,
    QgsServerQueryStringLengthParameter,
    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):
        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)
The code snippets on this page need the following imports if you’re outside the pyqgis console:

```python
from qgis.QtCore import  
 QRectF,
}

from qgis.core import 
 QgsProject,
QgsLayerTreeModel,
}

from qgis.gui import 
 QgsLayerTreeView,
}
```
20.1 User Interface

Change Look & Feel

```python
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())
```

Change icon and title

```python
from qgis.PyQt.QtGui import QIcon
icon = QIcon("/path/to/logo/file.png")
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("My QGIS")
```

20.2 설정

Get QgsSettings list

```python
from qgis.core import QgsSettings
qs = QgsSettings()
for k in sorted(qs.allKeys()):
    print (k)
```
## 20.3 툴바

### Remove toolbar

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

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

### Remove actions toolbar

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

## 20.4 Menus

### Remove menu

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

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

## 20.5 Canvas

### Access canvas

```
canvas = iface.mapCanvas()
```

### Change canvas color

```
from qgis.Qt import Qt
iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

### Map Update interval

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

Add vector layer

```python
layer = iface.addVectorLayer("testdata/airports.shp", "layer name you like", "ogr")
if not layer or not layer.isValid():
    print("Layer failed to load!")
```

Get active layer

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

List all layers

```python
from qgis.core import QgsProject
QgsProject.instance().mapLayers().values()
```

Obtain layers name

```python
from qgis.core import QgsVectorLayer
layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer)
layers_names = []
for layer in QgsProject.instance().mapLayers().values():
    layers_names.append(layer.name())
print("layers TOC = ", format(layers_names))
```

Otherwise

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

Find layer by name

```python
from qgis.core import QgsProject
layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
print(layer.name())
```

Set active layer

```python
from qgis.core import QgsProject
layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
iface.setActiveLayer(layer)
```

Refresh layer at interval
from qgis.core import QgsProject

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

# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)

# Enable auto refresh
layer.setAutoRefreshEnabled(True)

Show methods

dir(layer)

Adding new feature with feature form

from qgis.core import QgsFeature, QgsGeometry

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

iface.openFeatureForm(layer, feat, False)

Adding new feature without feature form

from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

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

Get features

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

Get selected features

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

Get selected features Ids

selected_ids = layer.selectedFeatureIds()
print(selected_ids)

Create a memory layer from selected features Ids

from qgis.core import QgsFeatureRequest

memory_layer = layer.materialize(QgsFeatureRequest().setFilterFids(layer._selectedFeatureIds()))
QgsProject.instance().addMapLayer(memory_layer)

Get geometry
# Point layer

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

10.000000, 10.000000

**Move geometry**

```python
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)>

**Set the CRS**

```python
from qgis.core import QgsProject, QgsCoordinateReferenceSystem
for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem('EPSG:4326'))
```

**See the CRS**

```python
from qgis.core import QgsProject
for layer in QgsProject.instance().mapLayers().values():
    crs = layer.crs().authid()
    layer.setName('/{} {}').format(layer.name(), crs)
```

**Hide a field column**

```python
from qgis.core import QgsEditorWidgetSetup

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

**Layer from WKT**

```python
from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject
layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.0934.89,-88.39 30.34,-89.57 30.18,-89.73 31,-91.63 30.99,-90.8732.37,-91.23 33.44,-90.93 34.23,-90.30 34.99,-88.82 -34.99))")
```

(다음 페이지에 계속)
Load all vector layers from GeoPackage

```python
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)
```

Load tile layer (XYZ-Layer)

```python
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')
```

Remove all layers

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

Remove all

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

20.7 Table of contents

Access checked layers

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

Remove contextual menu

```python
ltv = iface.layerTreeView()
mp = ltv.menuProvider()
ltv.setMenuProvider(\None)
# Restore
ltv.setMenuProvider(mp)
```
20.8 Advanced TOC

Root node

```python
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())
```

Access the first child node

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

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

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

Find groups and nodes

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

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

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

Find group by name

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

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

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

- group: My Group
  - layer: layer name you like
print (root.findGroup("My Group"))

<qgis._core.QgsLayerTreeGroup object at 0x7fd75560cee8>

Find layer by id

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

<qgis._core.QgsLayerTreeLayer object at 0x7f56087af288>

Add layer

```python
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)
```

Add group

```python
from qgis.core import QgsLayerTreeGroup

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

Move loaded layer

```python
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)
```

Move loaded layer to a specific group

```python
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(myLayer)
```

Changing visibility
myGroup.setItemVisibilityChecked(False)
myLayer.setItemVisibilityChecked(False)

Is group selected

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

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

Expand node

print(myGroup.isExpanded())
myGroup.setExpanded(False)

Hidden node trick

from qgis.core import QgsProject
model = iface.layerTreeView().layerTreeModel()
ltv = iface.layerTreeView()
root = QgsProject.instance().layerTreeRoot()
layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
note = root.findLayer( layer.id() )
index = model.node2index( note )
ltv.setRowHidden( index.row(), index.parent(), True )
note.setCustomProperty( 'nodeHidden', 'true' )
ltv.setCurrentIndex(model.node2index(root))

Node signals

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)

Remove layer

root.removeLayer(layer)

Remove group

root.removeChildNode(node_group2)

Create new table of contents (TOC)

root = QgsProject.instance().layerTreeRoot()
model = QgsLayerTreeModel(root)

(다음 페이지에 계속)
Move node

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

Rename node

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

## 20.9 Processing algorithms

### Get algorithms list

```python
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

### Get algorithms help

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

... 

### Run the algorithm

For this example, the result is stored in a temporary memory layer which is added to the project.

```python
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'}

### How many algorithms are there?

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

### How many providers are there?

```python
...
from qgis.core import QgsApplication
len(QgsApplication.processingRegistry().providers())

How many expressions are there?

from qgis.core import QgsExpression
len(QgsExpression.Functions())

## 20.10 Decorators

CopyRight

```python
from qgis.Qt import QTextDocument
from qgis.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...
    deviceWidth = p.device().width() # Get paint device width on which this painter...
    # Create new container for structured rich text
    text = QTextDocument()
    font = QFont()
    font.setFamily(mQFont)
    font.setPointSize(int(mQFontsize))
    textsetDefaultFont(font)
    style = "<style type="text/css">
    p { color: " + mLabelQColor + "}\</style>"

    # 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
```

(다음 페이지에 계속)
```python
40 if case == 0:
    # Top Left
    add_copyright(p, text, xOffset, yOffset)

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

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

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

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

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

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

## 20.11 Composer

Get print layout by name

```python
composerTitle = 'MyComposer' # Name of the composer
project = QgsProject.instance()
projectLayoutManager = project.layoutManager()
layout = projectLayoutManager.layoutByName(composerTitle)
```
20.12 Sources

- QGIS Python (PyQGIS) API
- QGIS C++ API
- StackOverflow QGIS questions
- Script by Klas Karlsson