PyQGIS 3.10 developer cookbook

QGIS Project

Dec 09, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

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.

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

A 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:

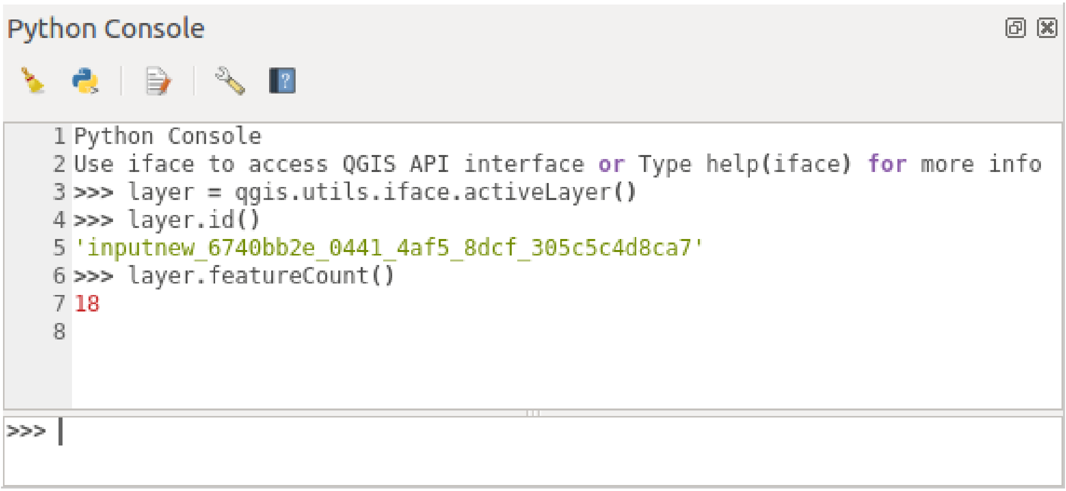


Fig. 1.1: QGIS Python console

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)

```
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 Python Plugins

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.

Creating plugins in Python is simple, see *Developing Python Plugins* for detailed instructions.

Note: Python plugins are also available for QGIS server. See *QGIS Server and Python* for further details.

1.3 Running Python code when QGIS starts

There are two distinct methods to run Python code every time QGIS starts.

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

1.3.1 The `startup.py` file

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.

Note: 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 undesireable paths, or for isolating/loading the initial environment without requiring a virtual environment, e.g. homebrew or MacPorts installs on Mac.

1.4 Python Applications

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.

Note: 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:

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

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.

```
from qgis.core import *

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

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

qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

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

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

(continues on next page)

(continued from previous page)

```
# 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 :-)

1.4.3 Running Custom Applications

You need to tell your system where to search for QGIS libraries and appropriate Python modules if they are not in a well-known location - otherwise Python will complain:

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

This can be fixed by setting the `PYTHONPATH` environment variable. In the following commands, `<qgispath>` should be replaced with your actual QGIS installation path:

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

Now, the path to the PyQGIS modules is known, but they depend on the `qgis_core` and `qgis_gui` libraries (the Python modules serve only as wrappers). The path to these libraries may be unknown to the operating system, and then you will get an import error again (the message might vary depending on the system):

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

Fix this by adding the directories where the QGIS libraries reside to the search path of the dynamic linker:

- on Linux: **`export LD_LIBRARY_PATH=/<qgispath>/lib`**
- on Windows: **`set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%`** where `<qgisrelease>` should be replaced with the type of release you are targeting (eg, `qgis-ltr`, `qgis`, `qgis-dev`)

These commands can be put into a bootstrap script that will take care of the startup. When deploying custom applications using PyQGIS, there are usually two possibilities:

- require the user to install QGIS prior to installing your application. The application installer should look for default locations of QGIS libraries and allow the user to set the path if not found. This approach has the advantage of being simpler, however it requires the user to do more steps.
- package QGIS together with your application. Releasing the application may be more challenging and the package will be larger, but the user will be saved from the burden of downloading and installing additional pieces of software.

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:

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

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

CHAPTER

TWO

LOADING PROJECTS

Sometimes you need to load an existing project from a plugin or (more often) when developing a standalone QGIS Python application (see: *Python Applications*).

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:

```
# 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
import os
print(os.getcwd())
project.read('testdata/01_project.qgs')
print(project.fileName())
```

```
...
testdata/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:

```
# 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.

Note: 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:

```
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:

```
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:

```
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:

```
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)
```

CHAPTER

THREE

LOADING LAYERS

The code snippets on this page need the following imports:

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

- *Vector Layers*
- *Raster Layers*
- *QgsProject instance*

Let's open some layers with data. QGIS recognizes vector and raster layers. Additionally, custom layer types are available, but we are not going to discuss them here.

3.1 Vector Layers

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:

```
# 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!")
else:
    QgsProject.instance().addMapLayer(vlayer)
```

The data source identifier is a string and it is specific to each vector data provider. Layer's name is used in the layer list widget. It is important to check whether the layer has been loaded successfully. If it was not, an invalid layer instance is returned.

For a geopackage vector layer:

```
# 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"
```

(continues on next page)

(continued from previous page)

```
# 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`:

```
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.

The following list shows how to access various data sources using vector data providers:

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

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

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

    ```
    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:

  ```
  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")
  ```

  Note: The `False` argument passed to `uri.uri(False)` prevents the expansion of the authentication configuration 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://{}/testdata/delimited_xy.csv?delimiter={}&xField={}&yField={}".format(os.getcwd(), ";", "x", "y")
  vlayer = QgsVectorLayer(uri, "layer name you like", "delimitedtext")
  ```

(continues on next page)

(continued from previous page)

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

Note: 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(";", "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:

```
uri = "https://demo.geo-solutions.it/geoserver/ows?service=WFS&version=1.1.0&
↪request=GetFeature&typename=geosolutions:regioni"
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")
QgsProject.instance().addMapLayer(vlayer)
```

The uri can be created using the standard `urllib` library:

```
import urllib

params = {
    'service': 'WFS',
    'version': '1.1.0',
    'request': 'GetFeature',
    'typename': 'geosolutions:regioni',
    'srsname': "EPSG:4326"
}
uri2 = 'https://demo.geo-solutions.it/geoserver/ows?' + urllib.parse.
↪unquote(urllib.parse.urlencode(params))
```

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

```
uri = "https://demo.geo-solutions.it/geoserver/ows?service=WFS&version=1.1.0&
↪request=GetFeature&typename=geosolutions:regioni"
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)
QgsProject.instance().addMapLayer(vlayer)
```

3.2 Raster Layers

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:

```
# 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:

```
# 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:

```
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 create a dictionary of strings for the database connection parameters. The dictionary is then loaded into an empty URI, before adding the raster. Note that None should be used when it is desired to leave the parameter blank:

```
uri_config = {#
# a dictionary of database parameters
'dbname':'gis_db', # The PostgreSQL database to connect to.
'host':'localhost',      # The host IP address or localhost.
'port':'5432',           # The port to connect on.
'sslmode':'disable',     # The SSL/TLS mode. Options: allow, disable, prefer,␣
↪require, verify-ca, verify-full
# user and password are not needed if stored in the authcfg or service
'user':None,             # The PostgreSQL user name, also accepts the new WFS␣
↪provider naming.
```

(continues on next page)

(continued from previous page)

```
'password':None,          # The PostgreSQL password for the user.
'service':None,           # The PostgreSQL service to be used for connection to the␣
↪database.
'authcfg':'QconfigId',  # The QGIS athentication database ID holding connection␣
↪details.
# table and raster column details
'schema':'public',        # The database schema that the table is located in.
'table':'my_rasters',   # The database table to be loaded.
'column':'rast',          # raster column in PostGIS table
'mode':'2',               # GDAL 'mode' parameter, 2 union raster tiles, 1 separate␣
↪tiles (may require user input)
'sql':None,               # An SQL WHERE clause.
'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.
'connect_timeout':None,
'hostaddr':None,
'driver':None,
'tty':None,
'requiressl':None,
'krbsrvname':None,
'gsslib':None,
}
# configure the URI string with the dictionary
uri = QgsDataSourceUri()
for param in uri_config:
    if (uri_config[param] != None):
        uri.setParam(param, uri_config[param]) # add parameters to the URI

# the raster can now be loaded into the project using the URI string and GDAL data␣
↪provider
rlayer = iface.addRasterLayer('PG: ' + uri.uri(False), "raster layer name", "gdal")
```

Raster layers can also be created from a WCS service:

```
layer_name = 'nurc:mosaic'
uri = "https://demo.geo-solutions.it/geoserver/ows?identifier={}".format(layer_
↪name)
rlayer = QgsRasterLayer(uri, 'my wcs layer', 'wcs')
```

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:

```
urlWithParams = "crs=EPSG:4326&format=image/png&layers=tasmania&styles&url=https://demo.geo-solutions.it/geoserver/ows"
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
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:

```
# 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:

```
# 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:

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

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

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

CHAPTER

FOUR

ACCESSING THE TABLE OF CONTENTS (TOC)

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:

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

```
{'countries_89ae1b0f_f41b_4f42_bca4_caf55ddbe4b6': <QgsMapLayer: '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:

```
# 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)
```

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

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

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

Note: 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).

Note: 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)
```

```
[<QgsMapLayer: '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:

```
# 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:

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

You can switch between QgsVectorLayer and QgsLayerTreeLayer easily:

```
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 0x7fecceb46ca8>
Map layer: <QgsMapLayer: '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:

```
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

```
# 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:

```
# 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:

```
# 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:

```
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:

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

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

CHAPTER

FIVE

USING RASTER LAYERS

5.1 Layer Details

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.

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

```
919 619
```

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

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

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

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

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

```
0
```

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

```
1
```

```
# get 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`):

- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`
- `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 Single Band Rasters

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:

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

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:

```
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:

```
rlayer.triggerRepaint()
```

5.2.2 Multi Band Rasters

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

```
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:

```
rlayer_multi.triggerRepaint()
```

5.3 Query Values

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:

```
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.

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

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

```
{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:

```
from qgis.core import (
  QgsApplication,
  QgsDataSourceUri,
  QgsCategorizedSymbolRenderer,
  QgsClassificationRange,
  QgsPointXY,
  QgsProject,
  QgsExpression,
  QgsField,
  QgsFields,
  QgsFeature,
```

(continues on next page)

(continued from previous page)

```
    QgsFeatureRequest,
    QgsFeatureRenderer,
    QgsGeometry,
    QgsGraduatedSymbolRenderer,
    QgsMarkerSymbol,
    QgsMessageLog,
    QgsRectangle,
    QgsRendererCategory,
    QgsRendererRange,
    QgsSymbol,
    QgsVectorDataProvider,
    QgsVectorLayer,
    QgsVectorFileWriter,
    QgsWkbTypes,
    QgsSpatialIndex,
)

from qgis.core.additions.edit import edit

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

CHAPTER

SIX

USING VECTOR LAYERS

This section summarizes various actions that can be done with vector layers.

Most work here is based on the methods of the `QgsVectorLayer` class.

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:

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

```
ID Integer64
fk_region Integer64
ELEV Real
NAME String
USE String
```

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:

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

```
NAME
```

Note: If you change the `Display Name` from a field to an expression, you have to use `displayExpression()` instead of `displayField()`.

6.2 Iterating over Vector Layer

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.

```
# "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
    # show some information about the feature geometry
    geom = feature.geometry()
    geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
    if geom.type() == QgsWkbTypes.PointGeometry:
        # the geometry type can be of single or multi type
        if geomSingleType:
            x = geom.asPoint()
            print("Point: ", x)
        else:
            x = geom.asMultiPoint()
            print("MultiPoint: ", x)
    elif geom.type() == QgsWkbTypes.LineGeometry:
        if geomSingleType:
```

(continues on next page)

(continued from previous page)

```
            x = geom.asPolyline()
            print("Line: ", x, "length: ", geom.length())
        else:
            x = geom.asMultiPolyline()
            print("MultiLine: ", x, "length: ", geom.length())
    elif geom.type() == QgsWkbTypes.PolygonGeometry:
        if geomSingleType:
            x = geom.asPolygon()
            print("Polygon: ", x, "Area: ", geom.area())
        else:
            x = geom.asMultiPolygon()
            print("MultiPolygon: ", x, "Area: ", geom.area())
    else:
        print("Unknown or invalid geometry")
    # fetch attributes
    attrs = feature.attributes()
    # attrs is a list. It contains all the attribute values of this feature
    print(attrs)
    # for this test only print the first feature
    break
```

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

6.3 Selecting features

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:

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

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

```
# 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:

```
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:

```
selected_fid = []

# Get the first feature id from the layer
```

(continues on next page)

(continued from previous page)

```
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:

```
layer.removeSelection()
```

6.3.1 Accessing attributes

Attributes can be referred to by their name:

```
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:

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

```
First feature
```

6.3.2 Iterating over selected features

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

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

6.3.3 Iterating over a subset of features

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:

```
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:

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

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

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

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

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:

```
# 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 *Expressions, Filtering and Calculating Values* for the details about the syntax supported by QgsExpression.

The request can be used to define the data retrieved for each feature, so the iterator returns all features, but returns partial data for each of them.

```
# 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 Modifying Vector Layers

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.

```
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:

```
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,
```

(continues on next page)

(continued from previous page)

```
# Presimplify Geometries, Presimplify Geometries with Validity Check,
# Transactions, Curved Geometries'
```

By using any of the following methods for vector layer editing, the changes are directly committed to the underlying data store (a file, database etc). In case you would like to do only temporary changes, skip to the next section that explains how to do *modifications with editing buffer*.

Note: 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:

```
# 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 Add Features

Create some `QgsFeature` instances and pass a list of them to provider's `addFeatures()` method. It will return two values: result (true/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.

```
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 Delete Features

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

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

6.4.3 Modify Features

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.

```
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:
```

(continues on next page)

(continued from previous page)

```
    geom = QgsGeometry.fromPointXY(QgsPointXY(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

Tip: 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.).

6.4.4 Modifying Vector Layers with an Editing Buffer

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.

```
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:

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

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

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

# ... more editing ...

layer.endEditCommand()
```

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:

```
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 Adding and Removing Fields

To add fields (attributes), you need to specify a list of field definitions. For deletion of fields just provide a list of field indexes.

```
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])
```

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

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

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

After adding or removing fields in the data provider the layer's fields need to be updated because the changes are not automatically propagated.

```
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 *Modifying Vector Layers with an Editing Buffer*.

6.5 Using Spatial Index

Spatial indexes can dramatically improve the performance of your code if you need to do frequent queries to a vector layer. Imagine, for instance, that you are writing an interpolation algorithm, and that for a given location you need to know the 10 closest points from a points layer, in order to use those point for calculating the interpolated value. Without a spatial index, the only way for QGIS to find those 10 points is to compute the distance from each and every point to the specified location and then compare those distances. This can be a very time consuming task, especially if it needs to be repeated for several locations. If a spatial index exists for the layer, the operation is much more effective.

Think of a layer without a spatial index as a telephone book in which telephone numbers are not ordered or indexed. The only way to find the telephone number of a given person is to read from the beginning until you find it.

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:

```
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.

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

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

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

- once spatial index is filled with some values, you can do some queries

```
# 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))
```

6.6 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`...).

6.6.1 From an instance of `QgsVectorFileWriter`

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

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "ESRI Shapefile"
save_options.fileEncoding = "UTF-8"
transform_context = QgsProject.instance().transformContext()
error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
                                                  "testdata/my_new_shapefile",
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
else:
  print(error)
```

```
# Write to an ESRI GDB file
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "FileGDB"
# if no geometry
save_options.overrideGeometryType = QgsWkbTypes.Unknown
save_options.actionOnExistingFile = QgsVectorFileWriter.CreateOrOverwriteLayer
save_options.layerName = 'my_new_layer_name'
transform_context = QgsProject.instance().transformContext()
gdb_path = "testdata/my_example.gdb"
error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
                                                gdb_path,
                                                transform_context,
                                                save_options)
if error[0] == QgsVectorFileWriter.NoError:
  print("success!")
else:
  print(error)
```

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:

```
LIST_FIELD_NAME = 'xxxx'

class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

  def __init__(self, layer, list_field):
    QgsVectorFileWriter.FieldValueConverter.__init__(self)
    self.layer = layer
```

(continues on next page)

(continued from previous page)

```
    self.list_field_idx = self.layer.fields().indexFromName(list_field)

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

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

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

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.

6.6.2 Directly from features

```
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(
```

(continues on next page)

(continued from previous page)

```
    "testdata/my_new_shapefile.shp",
    fields,
    QgsWkbTypes.Point,
    crs,
    transform_context,
    save_options
)

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

# add a feature
fet = QgsFeature()

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

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

6.6.3 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.

The provider supports string, int and double fields.

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"`.

The URI can also specify the coordinate reference system, fields, and indexing of the memory provider in the URI. The syntax is:

crs=definition Specifies the coordinate reference system, where definition may be any of the forms accepted by `QgsCoordinateReferenceSystem.createFromString`

index=yes Specifies that the provider will use a spatial index

field=name:type(length,precision) Specifies an attribute of the layer. The attribute has a name, and optionally a type (integer, double, or string), length, and precision. There may be multiple field definitions.

The following example of a URI incorporates all these options

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

The following example code illustrates creating and populating a memory provider

```
from qgis.PyQt.QtCore import QVariant

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

# add fields
```

(continues on next page)

(continued from previous page)

```
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()
```

Finally, let's check whether everything went well

```
# 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)>
```

6.7 Appearance (Symbology) of Vector Layers

When a vector layer is being rendered, the appearance of the data is given by **renderer** and **symbols** associated with the layer. Symbols are classes which take care of drawing of visual representation of features, while renderers determine what symbol will be used for a particular feature.

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

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

And with that reference, let us explore it a bit

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

```
Type: singleSymbol
```

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

Type	Class	Description
singleSymbol	QgsSingleSymbolRenderer	Renders all features with the same symbol
categorizedSymbol	QgsCategorizedSymbolRenderer	Renders features using a different symbol for each category
graduatedSymbol	QgsGraduatedSymbolRenderer	Renders features using a different symbol for each range of values

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:

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

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

It is possible to obtain a dump of a renderer contents in text form — can be useful for debugging

```
renderer.dump()
```

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

6.7.1 Single Symbol Renderer

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:

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

`name` indicates the shape of the marker, and can be any of the following:

- `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:0,0,0,0,0,0', '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:

```
# 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()
```

6.7.2 Categorized Symbol Renderer

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

To get a list of categories

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

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

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

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.

6.7.3 Graduated Symbol Renderer

This renderer is very similar to the categorized symbol renderer described above, but instead of one attribute value per class it works with ranges of values and thus can be used only with numerical attributes.

To find out more about ranges used in the renderer

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

(continues on next page)

(continued from previous page)

```

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

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

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.

If you wish to create your own graduated symbol renderer you can do so as illustrated in the example snippet below (which creates a simple two class arrangement)

```
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)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbol.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setOpacity(myOpacity)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
↪"EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
```

6.7.4 Working with Symbols

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:

```
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.

Size and width are in millimeters by default, angles are in degrees.

Working with Symbol Layers

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:

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

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

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.

Creating Custom Symbol Layer Types

Imagine you would like to customize the way how the data gets rendered. You can create your own symbol layer class that will draw the features exactly as you wish. Here is an example of a marker that draws red circles with specified radius

```
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: `startRender` is called before rendering the first feature, `stopRender` when the rendering is done, and `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 `renderPolyline` which receives a list of lines, while `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.

Usually it is convenient to add a GUI for setting attributes of the symbol layer type to allow users to customize the appearance: in case of our example above we can let user set circle radius. The following code implements such widget

```
from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

        # setup a simple UI
        self.label = QLabel("Radius:")
```

(continues on next page)

(continued from previous page)

```
        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 `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 `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 `changed()` signal to let the properties dialog update the symbol preview.

Now we are missing only the final glue: to make QGIS aware of these new classes. This is done by adding the symbol layer to registry. It is possible to use the symbol layer also without adding it to the registry, but some functionality will not work: e.g. loading of project files with the custom symbol layers or inability to edit the layer's attributes in GUI.

We will have to create metadata for the symbol layer

```
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)

QgsApplication.symbolLayerRegistry().addSymbolLayerType(FooSymbolLayerMetadata())
```

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.

The last step is to add this symbol layer to the registry — and we are done.

6.7.5 Creating Custom Renderers

It might be useful to create a new renderer implementation if you would like to customize the rules how to select symbols for rendering of features. Some use cases where you would want to do it: symbol is determined from a combination of fields, size of symbols changes depending on current scale etc.

The following code shows a simple custom renderer that creates two marker symbols and chooses randomly one of them for every feature

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


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

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

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

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

  def usedAttributes(self, context):
    return []

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

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

```
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())
```

(continues on next page)

(continued from previous page)

```
    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.

The last missing bit is the renderer metadata and registration in registry, otherwise loading of layers with the renderer will not work and user will not be able to select it from the list of renderers. Let us finish our RandomRenderer example

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

class RandomRendererMetadata(QgsRendererAbstractMetadata):

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

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

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

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

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

```
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).

6.8 Further Topics

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:

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

CHAPTER

SEVEN

GEOMETRY HANDLING

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

Sometimes one geometry is actually a collection of simple (single-part) geometries. Such a geometry is called a multi-part geometry. If it contains just one type of simple geometry, we call it multi-point, multi-linestring or multi-polygon. For example, a country consisting of multiple islands can be represented as a multi-polygon.

The coordinates of geometries can be in any coordinate reference system (CRS). When fetching features from a layer, associated geometries will have coordinates in CRS of the layer.

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

7.1 Geometry Construction

PyQGIS provides several options for creating a geometry:

- from coordinates

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

  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.

  Multi-part geometries go one level further: multi-point is a list of points, multi-linestring is a list of linestrings and multi-polygon is a list of polygons.

- from well-known text (WKT)

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

- from well-known binary (WKB)

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

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

7.2 Access to Geometry

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.

```
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.

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

```
Point
LineString
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:

```
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)>]]
```

Note: 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()`.

7.3 Geometry Predicates and Operations

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 \'Zu%\''
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())
```

```
Zubin Potok
Area:  0.040717371293465573
Perimeter:  0.9406133328077781
Zulia
Area:  3.708060762610232
Perimeter:  17.172123598311487
Zuid-Holland
Area:  0.4204687950359031
Perimeter:  4.098878517120812
Zug
Area:  0.027573510374275363
Perimeter:  0.7756605461489624
```

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 \'Zu%\''
features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))

for f in features:
```

(continues on next page)

(continued from previous page)

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

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

```
Zubin Potok
Perimeter (m): 87581.40256396442
Area (m2): 369302069.18814206
Area (km2): 369.30206918814207
Zulia
Perimeter (m): 1891227.0945423362
Area (m2): 44973645460.19726
Area (km2): 44973.64546019726
Zuid-Holland
Perimeter (m): 331941.8000214341
Area (m2): 3217213408.4100943
Area (km2): 3217.213408410094
Zug
Perimeter (m): 67440.22483063207
Area (m2): 232457391.52097562
Area (km2): 232.45739152097562
```

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,
# lets see how far is Tenerife from his home
santa = QgsPointXY(25.847899, 66.543456)
tenerife = QgsPointXY(-16.5735, 28.0443)

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

You can find many example of algorithms that are included in QGIS and use these methods to analyze and transform vector data. Here are some links to the code of a few of them.

- 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:

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

CHAPTER

EIGHT

PROJECTIONS SUPPORT

8.1 Coordinate reference systems

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

- specify CRS by its ID

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

  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.

- specify CRS by its well-known text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],' \
      'PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],' \
      'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
assert 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.

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

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

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

Output:

```
QGIS CRS ID: 3452
PostGIS SRID: 4326
Description: WGS 84
Projection Acronym: longlat
Ellipsoid Acronym: WGS84
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.

```
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(832713.79873844375833869 553423.
↪98688333143945783)>
Transformed back: <QgsPointXY: POINT(18 5)>
```

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

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

from qgis.PyQt.QtCore import Qt, QRectF

from qgis.core import (
```

(continues on next page)

(continued from previous page)

```
    QgsVectorLayer,
    QgsPoint,
    QgsPointXY,
    QgsProject,
    QgsGeometry,
    QgsMapRendererJob,
)

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

CHAPTER

NINE

USING THE MAP CANVAS

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.

To summarize, the map canvas architecture consists of three concepts:

- map canvas — for viewing of the map
- map canvas items — additional items that can be displayed on the map canvas
- map tools — for interaction with the map canvas

9.1 Embedding Map Canvas

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

```
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.

By default, map canvas has black background and does not use anti-aliasing. To set white background and enable anti-aliasing for smooth rendering

```
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.

```
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])
```

After executing these commands, the canvas should show the layer you have loaded.

9.2 Rubber Bands and Vertex Markers

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:

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

To show a polygon

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

Note that points for polygon is not a plain list: in fact, it is a list of rings containing linear rings of the polygon: first ring is the outer border, further (optional) rings correspond to holes in the polygon.

Rubber bands allow some customization, namely to change their color and line width

```
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

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

(in C++ it's possible to just delete the item, however in Python `del r` would just delete the reference and the object will still exist as it is owned by the canvas)

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:

```
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

```
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 Using Map Tools with Canvas

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.

```
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)
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

        self.pan()

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

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

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

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

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

9.4 Writing Custom Map Tools

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.

Here is an example of a map tool that allows to define a rectangular extent by clicking and dragging on the canvas. When the rectangle is defined, it prints its boundary coordinates in the console. It uses the rubber band elements described before to show the selected rectangle as it is being defined.

```
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()
```

(continues on next page)

(continued from previous page)

```

  def reset(self):
    self.startPoint = self.endPoint = None
    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 Writing Custom Map Canvas Items

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

```
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:

```
import os

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

(continues on next page)

(continued from previous page)

```
)

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

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

CHAPTER

TEN

MAP RENDERING AND PRINTING

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 Simple Rendering

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.

Here's an example:

```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)

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)

render.start()
```

10.2 Rendering layers with different CRS

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

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 — this item tells the libraries where to put the map itself. Here we create a map and stretch it over the whole paper size

```
map = QgsLayoutItemMap(layout)
layout.addItem(map)
```

- label — allows displaying labels. It is possible to modify its font, color, alignment and margin

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

- legend

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

- scale bar

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

- arrow

- picture
- basic shape
- nodes based shape

```
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.addItem(polygonItem)

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

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

- table

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

```
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.

```
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:

```
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:

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

CHAPTER

ELEVEN

EXPRESSIONS, FILTERING AND CALCULATING VALUES

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.

Three basic types are supported:

- number — both whole numbers and decimal numbers, e.g. `123`, `3.14`
- string — they have to be enclosed in single quotes: `'hello world'`
- column reference — when evaluating, the reference is substituted with the actual value of the field. The names are not escaped.

The following operations are available:

- arithmetic operators: `+`, `-`, `*`, `/`, `^`
- parentheses: for enforcing the operator precedence: `(1 + 1) * 3`
- unary plus and minus: `-12`, `+5`
- mathematical functions: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- conversion functions: `to_int`, `to_real`, `to_string`, `to_date`
- geometry functions: `$area`, `$length`
- geometry handling functions: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

And the following predicates are supported:

- comparison: `=`, `!=`, `>`, `>=`, `<`, `<=`
- pattern matching: `LIKE` (using % and _), `~` (regular expressions)
- logical predicates: `AND`, `OR`, `NOT`
- NULL value checking: `IS NULL`, `IS NOT NULL`

Examples of predicates:

- `1 + 2 = 3`
- `sin(angle) > 0`

- `'Hello' LIKE 'He%'`
- `(x > 10 AND y > 10) OR z = 0`

Examples of scalar expressions:

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

11.1 Parsing Expressions

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

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

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

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

11.2 Evaluating Expressions

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 Basic Expressions

This basic expression evaluates to 1, meaning it is true:

```
exp = QgsExpression('1 + 1 = 2')
assert(exp.evaluate()) # exp.evaluate() returns 1 and assert() recognizes this as␣
↪True
```

11.2.2 Expressions with features

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.

```
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)
assert(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:

```
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)

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

The following example can be used to filter a layer and return any feature that matches a predicate.

```
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

assert(matches == 7)
```

11.3 Handling expression errors

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

```
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:

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

CHAPTER

TWELVE

READING AND STORING SETTINGS

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.

```
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.

```
proj = QgsProject.instance()

# store values
proj.writeEntry("myplugin", "mytext", "hello world")
proj.writeEntry("myplugin", "myint", 10)
```

(continues on next page)

(continued from previous page)

```
proj.writeEntry("myplugin", "mydouble", 0.01)
proj.writeEntry("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 all data types, 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.

```
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:

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

from qgis.gui import (
    QgsMessageBar,
)

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

CHAPTER

THIRTEEN

COMMUNICATING WITH THE USER

This section shows some methods and elements that should be used to communicate with the user, in order to keep consistency in the User Interface.

13.1 Showing messages. The QgsMessageBar class

Using message boxes can be a bad idea from a user experience point of view. For showing a small info line or a warning/error messages, the QGIS message bar is usually a better option.

Using the reference to the QGIS interface object, you can show a message in the message bar with the following code

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

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

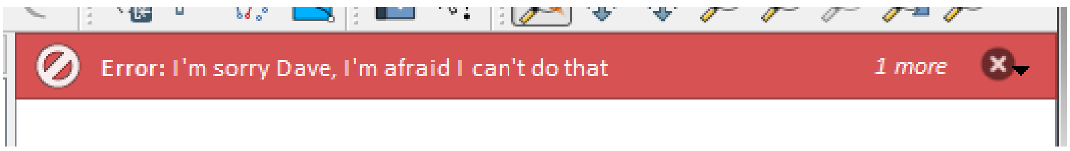


Fig. 13.1: QGIS Message bar

You can set a duration to show it for a limited time

```
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

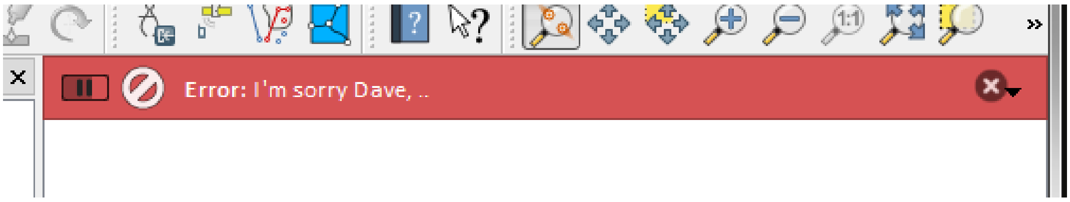


Fig. 13.2: QGIS Message bar with timer

1. Warning
2. Critical
3. Success

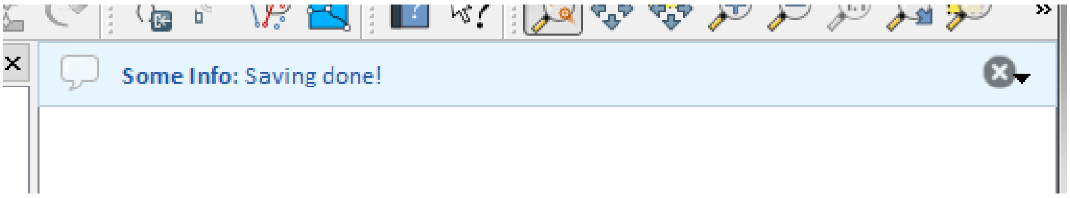


Fig. 13.3: QGIS Message bar (info)

Widgets can be added to the message bar, like for instance a button to show more info

```
def showError():
    pass

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

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

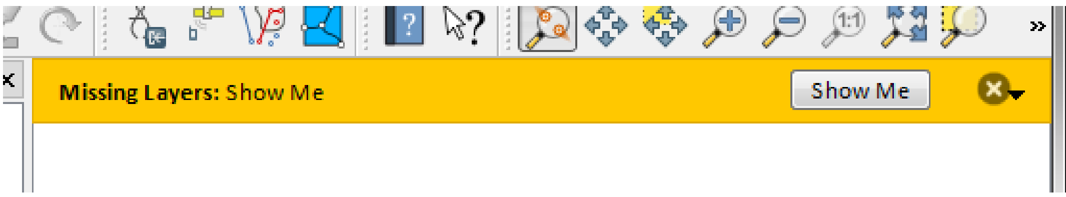


Fig. 13.4: QGIS Message bar with a button

You can even use a message bar in your own dialog so you don't have to show a message box, or if it doesn't make sense to show it in the main QGIS window

```
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)
```

(continues on next page)

(continued from previous page)

```
        self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
        self.layout().addWidget(self.bar, 0, 0, 1, 1)
    def run(self):
        self.bar.pushMessage("Hello", "World", level=Qgis.Info)

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

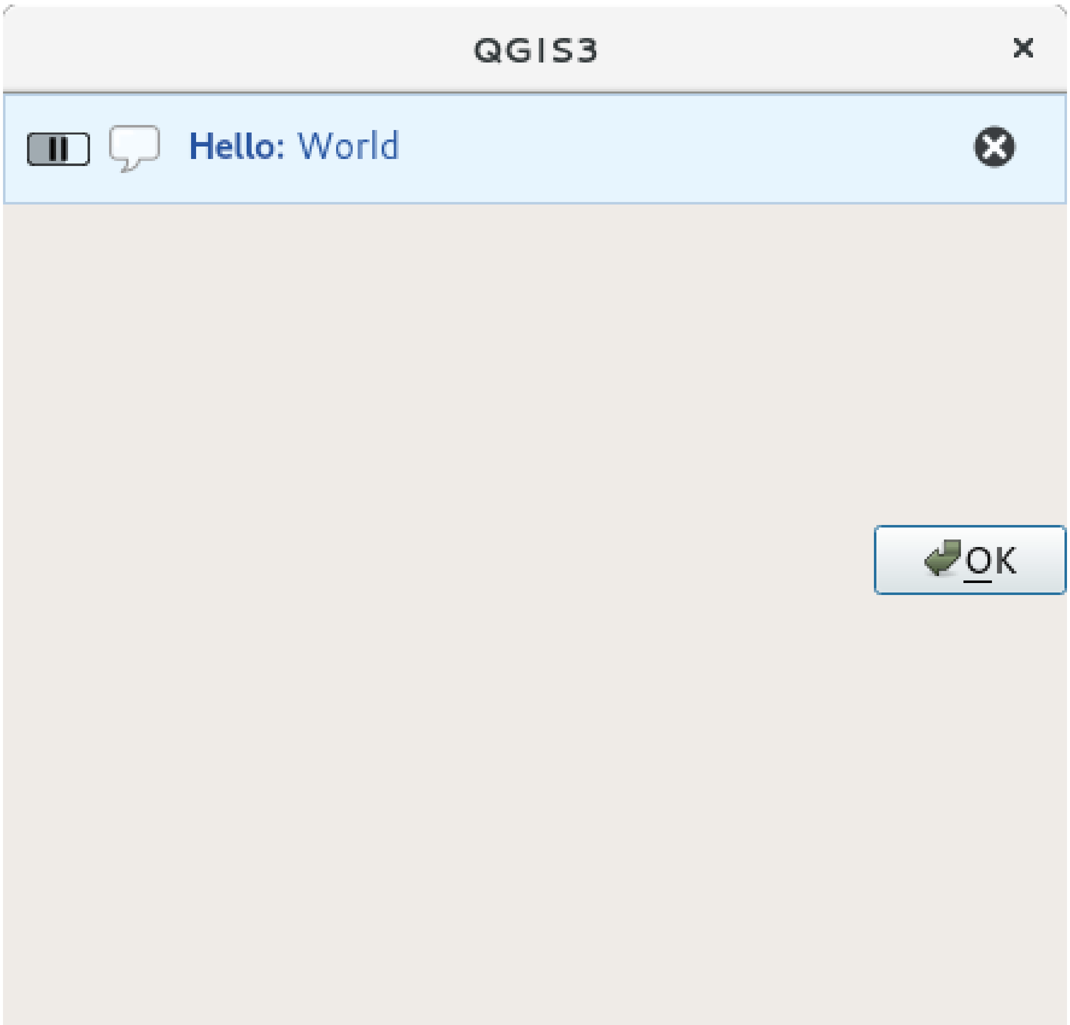


Fig. 13.5: QGIS Message bar in custom dialog

13.2 Showing progress

Progress bars can also be put in the QGIS message bar, since, as we have seen, it accepts widgets. Here is an example that you can try in the console.

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

(continues on next page)

(continued from previous page)

```
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)

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

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

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

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

```
vlayer = iface.activeLayer()

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

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

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

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

13.3 Logging

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** 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.

Warning: 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**, **renderers**, **symbol layers** and **Processing algorithms** (amongst others). In these cases you should always use the python **logging** module or thread safe classes (`QgsLogger` or `QgsMessageLog`) instead.

13.3.1 QgsMessageLog

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

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

Note: You can see the output of the `QgsMessageLog` in the log_message_panel

13.3.2 The python built in logging module

```
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.

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

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

```
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

Warning: 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:

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

(continues on next page)

(continued from previous page)

```
  QgsPkiBundle,
  QgsMessageLog,
)

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

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

from qgis.PyQt.QtNetwork import QSslCertificate
```

CHAPTER

FOURTEEN

AUTHENTICATION INFRASTRUCTURE

14.1 Introduction

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.

> **Warning:** 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* crypted sqlite db `qgis-auth.db` where *Authentication Configuration* are stored. e.g user/password, personal certificates and keys, Certificate Authorities

Authentication DB *Authentication Database*

Authentication Configuration 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.

```
authMgr = QgsApplication.authManager()

# check if QgsAuthManager has already been initialized... a side effect
# of the QgsAuthManager.init() is that AuthDbPath is set.
# QgsAuthManager.init() is executed during QGIS application init and hence
# you do not normally need to call it directly.
if authMgr.authenticationDatabasePath():
    # already initilised => we are inside a QGIS app.
    if authMgr.masterPasswordIsSet():
        msg = 'Authentication master password not recognized'
        assert authMgr.masterPasswordSame("your master password"), msg
    else:
        msg = 'Master password could not be set'
        # The verify parameter check 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
```

(continues on next page)

(continued from previous page)

```
    # 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:

```
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 informations will be stored. Hereafter an useful snippet to store PKI-path credentials for an hypothetic alice user:

```
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()

# 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

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

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:

```
# 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.

14.3.3 Remove an entry from authdb

We can remove an entry from *Authentication Database* using it's `authcfg` identifier with the following snippet:

```
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.

Note: 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:

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

Sample output:

```
['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:

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

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

> **Warning:** 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:

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

Note: 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:

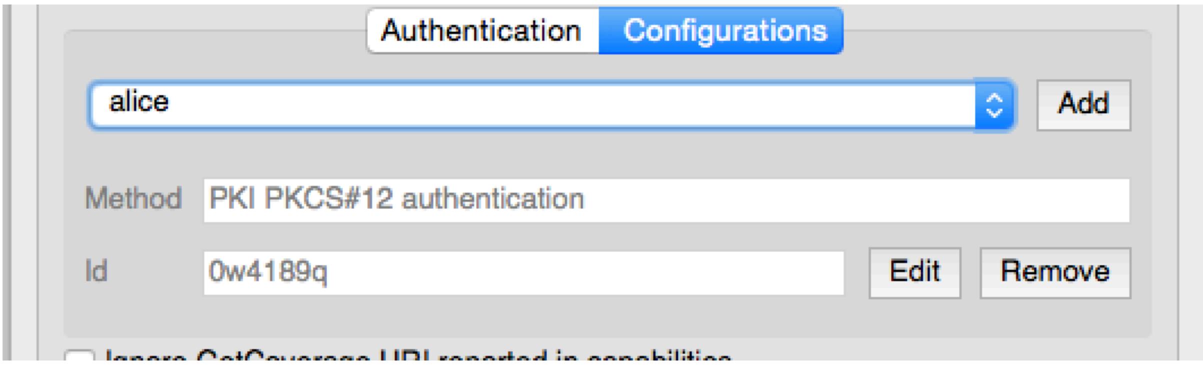
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

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 Method*s with the specified provider.

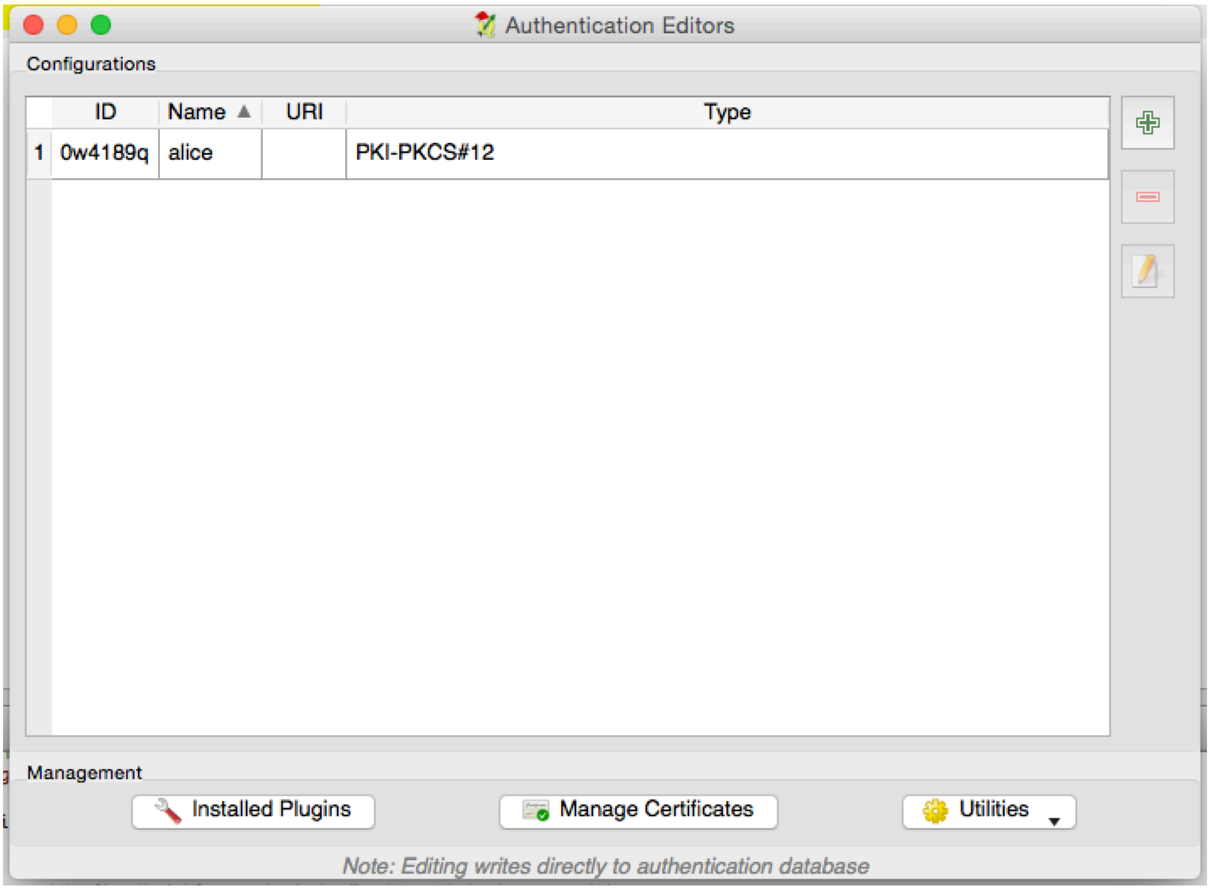
14.5.2 Authentication Editor GUI

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

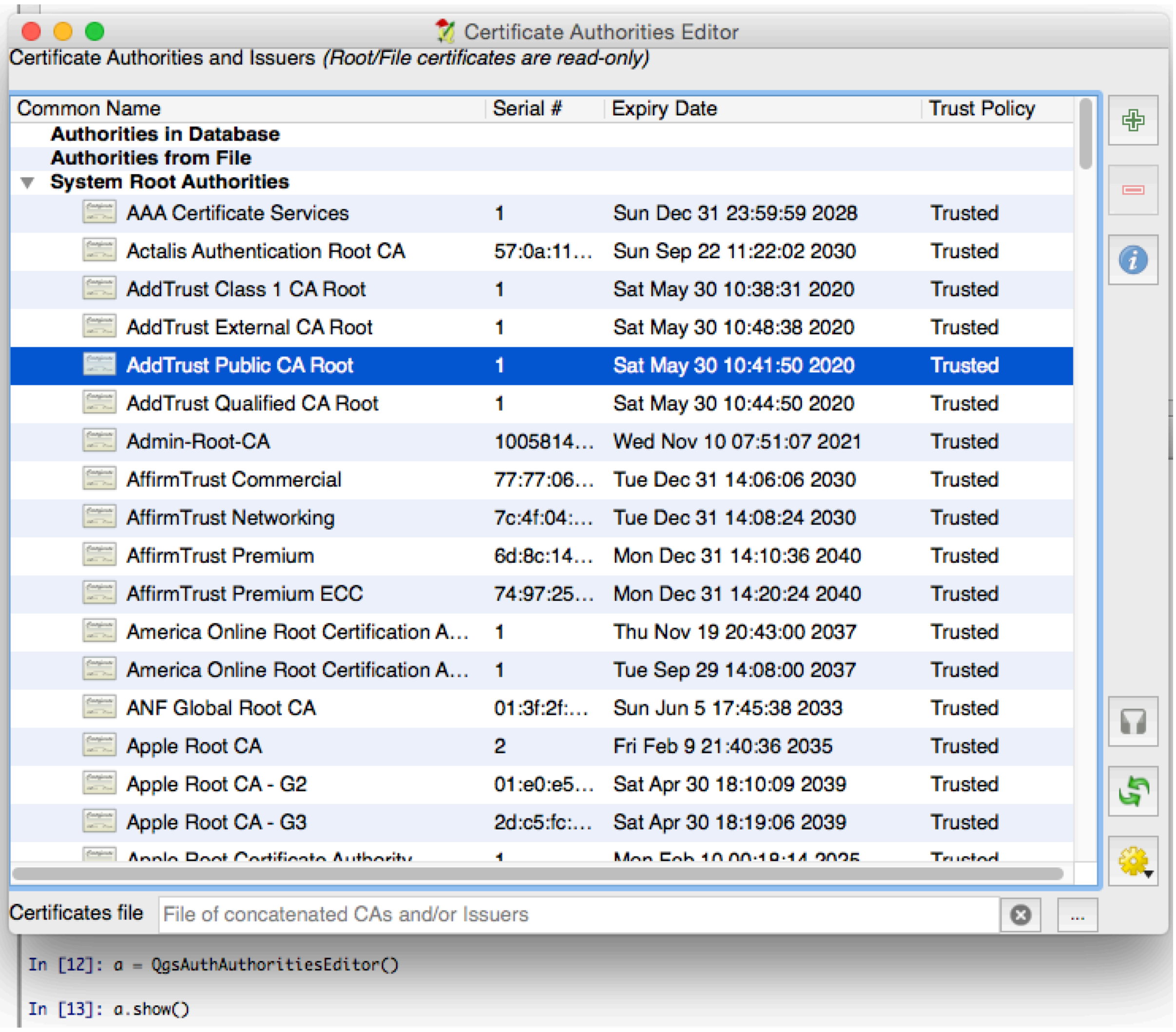
A GUI used to manage only authorities is managed by the `QgsAuthAuthoritiesEditor` class.

and can be used as in the following snippet:

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

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

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


Certificate Authorities Editor
Certificate Authorities and Issuers (Root/File certificates are read-only)
Common Name
Serial #
Expiry Date
Trust Policy
Authorities in Database
Authorities from File
System Root Authorities
AAA Certificate Services 1 Sun Dec 31 23:59:59 2028 Trusted
Actalis Authentication Root CA 57:0a:11… Sun Sep 22 11:22:02 2030 Trusted
AddTrust Class 1 CA Root 1 Sat May 30 10:38:31 2020 Trusted
AddTrust External CA Root 1 Sat May 30 10:48:38 2020 Trusted
AddTrust Public CA Root 1 Sat May 30 10:41:50 2020 Trusted
AddTrust Qualified CA Root 1 Sat May 30 10:44:50 2020 Trusted
Admin-Root-CA 1005814… Wed Nov 10 07:51:07 2021 Trusted
AffirmTrust Commercial 77:77:06… Tue Dec 31 14:06:06 2030 Trusted
AffirmTrust Networking 7c:4f:04:… Tue Dec 31 14:08:24 2030 Trusted
AffirmTrust Premium 6d:8c:14… Mon Dec 31 14:10:36 2040 Trusted
AffirmTrust Premium ECC 74:97:25… Mon Dec 31 14:20:24 2040 Trusted
America Online Root Certification A… 1 Thu Nov 19 20:43:00 2037 Trusted
America Online Root Certification A… 1 Tue Sep 29 14:08:00 2037 Trusted
ANF Global Root CA 01:3f:2f:… Sun Jun 5 17:45:38 2033 Trusted
Apple Root CA 2 Fri Feb 9 21:40:36 2035 Trusted
Apple Root CA - G2 01:e0:e5… Sat Apr 30 18:10:09 2039 Trusted
Apple Root CA - G3 2d:c5:fc:… Sat Apr 30 18:19:06 2039 Trusted
Certificates file
File of concatenated CAs and/or Issuers
...
In [12]: a = QgsAuthAuthoritiesEditor()
In [13]: a.show()

CHAPTER

FIFTEEN

TASKS - DOING HEAVY WORK IN THE BACKGROUND

15.1 Introduction

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`

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

- Create a task from a function

```
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)
```

Warning: 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 Examples

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.

```
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"""

    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)
```

(continues on next page)

(continued from previous page)

```
        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 "{name}" completed\n' \
                'RandomTotal: {total} (with {iterations} '\
              'iterations)'.format(
                  name=self.description(),
                  total=self.total,
                  iterations=self.iterations),
              MESSAGE_CATEGORY, Qgis.Success)
        else:
            if self.exception is None:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" not successful but without '\
                    'exception (probably the task was manually '\
                    'canceled by the user)'.format(
                        name=self.description()),
                    MESSAGE_CATEGORY, Qgis.Warning)
            else:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" Exception: {exception}'.format(
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception

    def cancel(self):
        QgsMessageLog.logMessage(
            'RandomTask "{name}" was canceled'.format(
                name=self.description()),
            MESSAGE_CATEGORY, Qgis.Info)
        super().cancel()

```

(continues on next page)

(continued from previous page)

```

longtask = RandomIntegerSumTask('waste cpu long', 20)
shorttask = RandomIntegerSumTask('waste cpu short', 10)
minitask = RandomIntegerSumTask('waste cpu mini', 5)
shortsubtask = RandomIntegerSumTask('waste cpu subtask short', 5)
longsubtask = RandomIntegerSumTask('waste cpu subtask long', 10)
shortestsubtask = RandomIntegerSumTask('waste cpu subtask shortest', 4)

# Add a subtask (shortsubtask) to shorttask that must run after
# minitask and longtask has finished
shorttask.addSubTask(shortsubtask, [minitask, longtask])
# Add a subtask (longsubtask) to longtask that must be run
# before the parent task
longtask.addSubTask(longsubtask, [], QgsTask.ParentDependsOnSubTask)
# Add a subtask (shortestsubtask) to longtask
longtask.addSubTask(shortestsubtask)

QgsApplication.taskManager().addTask(longtask)
QgsApplication.taskManager().addTask(shorttask)
QgsApplication.taskManager().addTask(minitask)
```

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

15.2.2 Task 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`.

```
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

def doSomething(task, wait_time):
    """
    Raises an exception to abort the task.
    Returns a result if success.
    The result will be passed, together with the exception (None in
    the case of success), to the on_finished method.
    If there is an exception, there will be no result.
    """
    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
                             MESSAGE_CATEGORY, Qgis.Info)
    wait_time = wait_time / 100
    total = 0
```

(continues on next page)

(continued from previous page)

```
    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()),
        MESSAGE_CATEGORY, Qgis.Info)

def completed(exception, result=None):
    """This is called when doSomething is finished.
    Exception is not None if doSomething raises an exception.
    result is the return value of doSomething."""
    if exception is None:
        if result is None:
            QgsMessageLog.logMessage(
                'Completed with no exception and no result '\
                '(probably manually canceled by the user)',
                MESSAGE_CATEGORY, Qgis.Warning)
        else:
            QgsMessageLog.logMessage(
                'Task {name} completed\n'
                'Total: {total} ( with {iterations} '
                'iterations)'.format(
                    name=result['task'],
                    total=result['total'],
                    iterations=result['iterations']),
                MESSAGE_CATEGORY, Qgis.Info)
    else:
        QgsMessageLog.logMessage("Exception: {}".format(exception),
                                 MESSAGE_CATEGORY, Qgis.Critical)
        raise exception

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

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

(continues on next page)

(continued from previous page)

```
RandomTaskFromFunction(0): Task Waste cpu 1 completed
RandomTotal: 25044 ( with 100 iterations)
```

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.

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

MESSAGE_CATEGORY = 'AlgRunnerTask'

def task_finished(context, successful, results):
    if not successful:
        QgsMessageLog.logMessage('Task finished unsucessfully',
                                 MESSAGE_CATEGORY, Qgis.Warning)
    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/.

CHAPTER

SIXTEEN

DEVELOPING PYTHON PLUGINS

16.1 Structuring Python Plugins

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*. 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

```
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.

> **Warning:** 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.

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

Metadata name	Required	Notes
name	True	a short string containing the name of the plugin
qgisMinimumVersion	True	dotted notation of minimum QGIS version
qgisMaximumVersion	False	dotted notation of maximum QGIS version
description	True	short text which describes the plugin, no HTML allowed
about	True	longer text which describes the plugin in details, no HTML allowed
version	True	short string with the version dotted notation
author	True	author name
email	True	email of the author, only shown on the website to logged in users, but visible in the Plugin Manager after the plugin is installed
changelog	False	string, can be multiline, no HTML allowed
experimental	False	boolean flag, *True* or *False* - *True* if this version is experimental
deprecated	False	boolean flag, *True* or *False*, applies to the whole plugin and not just to the uploaded version
tags	False	comma separated list, spaces are allowed inside individual tags
homepage	False	a valid URL pointing to the homepage of your plugin
repository	True	a valid URL for the source code repository
tracker	False	a valid URL for tickets and bug reports
icon	False	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)
category	False	one of *Raster*, *Vector*, *Database* and *Web*
plugin_dependencies	False	PIP-like comma separated list of other plugins to install
server	False	boolean flag, *True* or *False*, determines if the plugin has a server interface
hasProcessingProvider	False	boolean flag, *True* or *False*, determines if the plugin provides processing algorithms

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`

```
category=Raster
```

Note: If *qgisMaximumVersion* 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

```
; the next section is mandatory

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

; start of optional metadata
category=Raster
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

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

; these metadata can be empty, they will eventually become mandatory.
homepage=https://www.itopen.it
icon=icon.png

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

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

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=3.99

; Since QGIS 3.8, a comma separated list of plugins to be installed
; (or upgraded) can be specified.
; The example below will try to install (or upgrade) "MyOtherPlugin" version 1.12
; and any version of "YetAnotherPlugin"
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.PyQt.QtGui import *
from qgis.PyQt.QtWidgets import *

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

class TestPlugin:

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

  def initGui(self):
    # create action that will start plugin configuration
    self.action = QAction(QIcon(":/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):
```

(continues on next page)

(continued from previous page)

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

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

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
```

Note: In Windows environments, attempting to run the **pyrcc5** from Command Prompt or Powershell will probably result in the error "Windows cannot access the specified device, path, or file [...]". The easiest solution is probably to use the OSGeo4W Shell but if you are comfortable modifying the PATH environment variable or specifiying the path to the executable explicitly you should be able to find it at `<Your QGIS Install Directory>\bin\pyrcc5.exe`.

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.

> **Warning:** 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...)

.ts file

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.

> **Warning:** 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.

```
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 Code Snippets

This section features code snippets to facilitate plugin development.

16.2.1 How to call a method by a key shortcut

In the plug-in add to the `initGui()`

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

To `unload()` add

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

The method that is called when CTRL+I is pressed

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

16.2.2 How to toggle Layers

There is an API to access layers in the legend. Here is an example that toggles the visibility of the active layer

```
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 How to access attribute table of selected features

```
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 Interface for plugin in the options dialog

You can add a custom plugin options tab to *Settings* ▸ *Options*. This is preferable over adding a specific main menu entry for your plugin's options, as it keeps all of the QGIS application settings and plugin settings in a single place which is easy for users to discover and navigate.

The following snippet will just add a new blank tab for the plugin's settings, ready for you to populate with all the options and settings specific to your plugin. You can split the following classes into different files. In this example, we are adding two classes into the main `mainPlugin.py` file.

```
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)
```

Finally we are adding the imports and modifying the `__init__` function:

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

(continues on next page)

(continued from previous page)

```


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

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

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


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

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

Tip: You can apply a similar logic to add the plugin custom option to the layer properties dialog using the classes `QgsMapLayerConfigWidgetFactory` and `QgsMapLayerConfigWidget`.

16.3 Using Plugin Layers

If your plugin uses its own methods to render a map layer, writing your own layer type based on `QgsPluginLayer` might be the best way to implement that.

16.3.1 Subclassing QgsPluginLayer

Below is an example of a minimal QgsPluginLayer implementation. It is based on the original code of the Watermark example plugin.

The custom renderer is the part of the implement that defines the actual drawing on the canvas.

```
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):

```

(continues on next page)

(continued from previous page)

```
    LAYER_TYPE="watermark"

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

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

    def setTransformContext(self, ct):
        pass

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

    def readXml(self, node, context):
        pass

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

The plugin layer can be added to the project and to the canvas as any other map layer:

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

When loading a project containing such a layer, a factory class is needed:

```
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

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.

> **Warning:** *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 OSGeoW install

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

5. 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

Installation

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.

```
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.

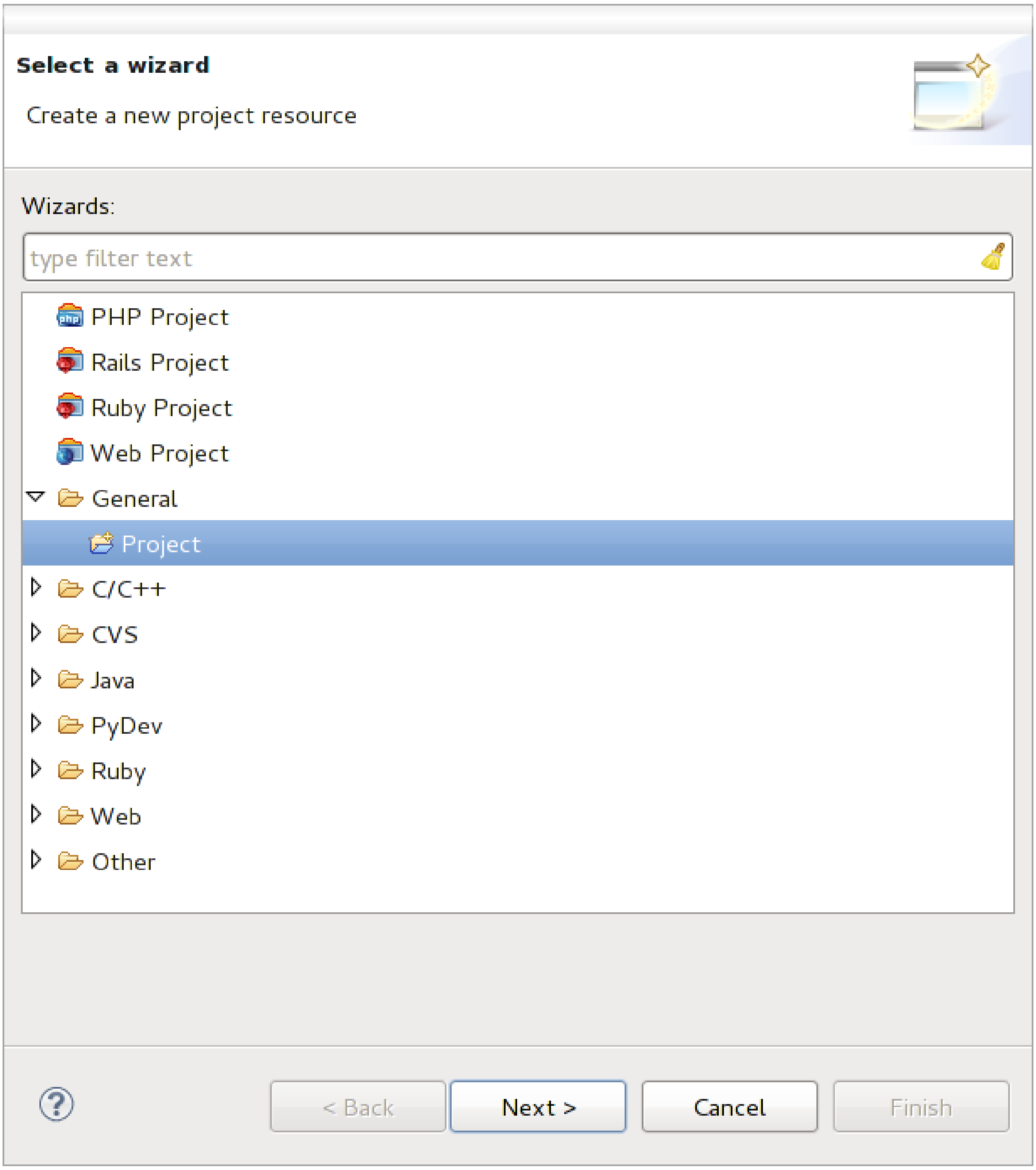


Fig. 16.1: Eclipse project

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.

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).

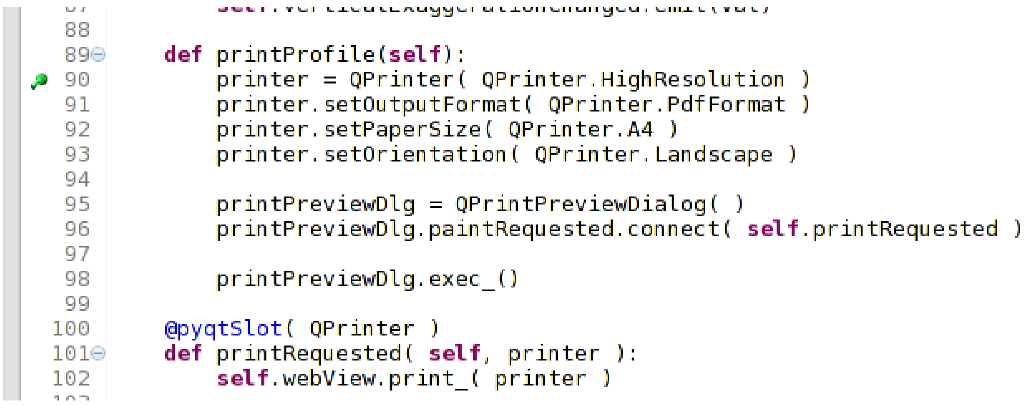


Fig. 16.2: Breakpoint

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.

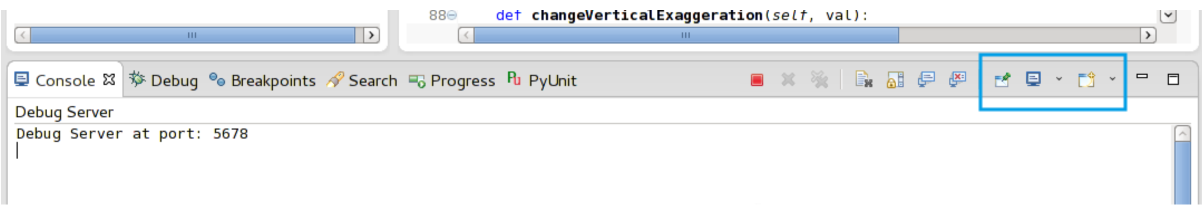


Fig. 16.3: PyDev Debug Console

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*.

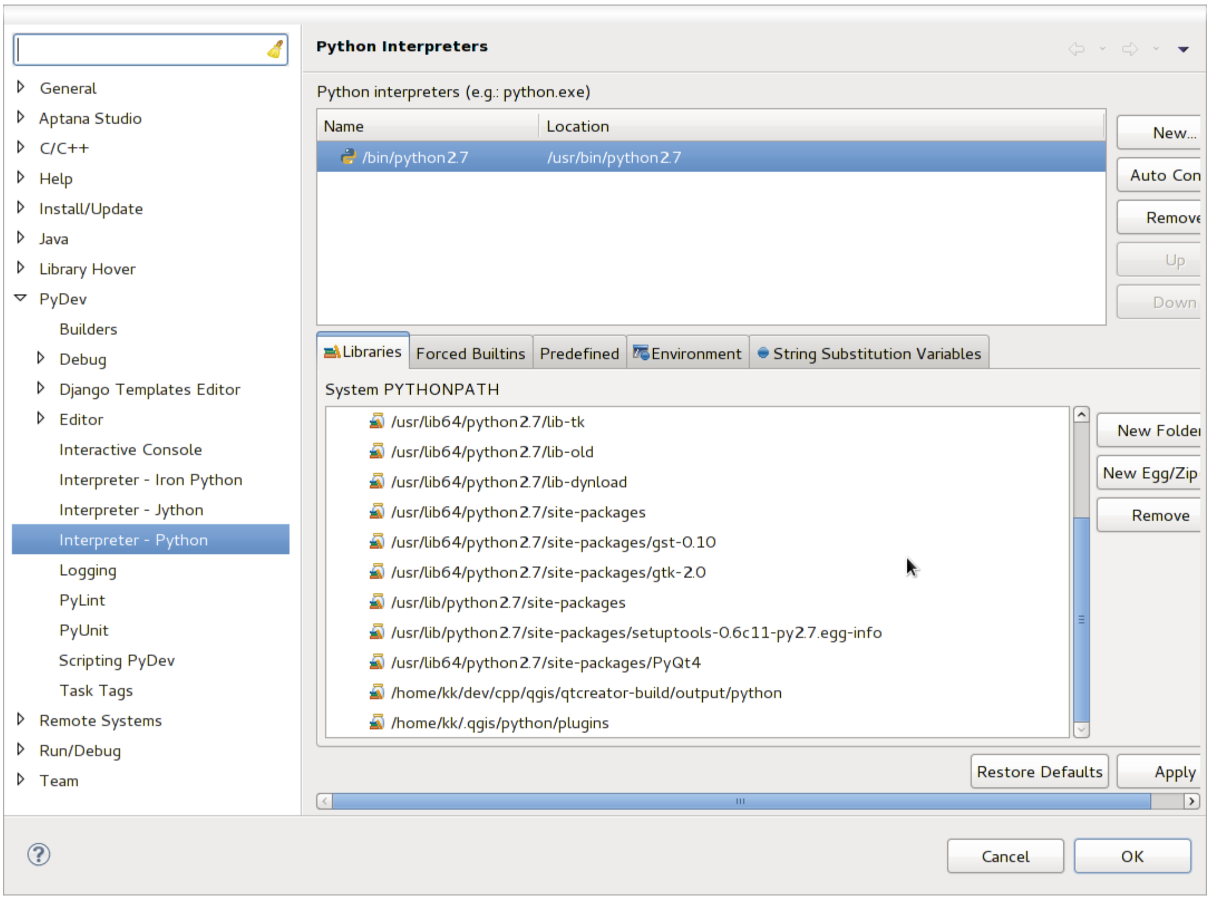


Fig. 16.4: PyDev Debug Console

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.

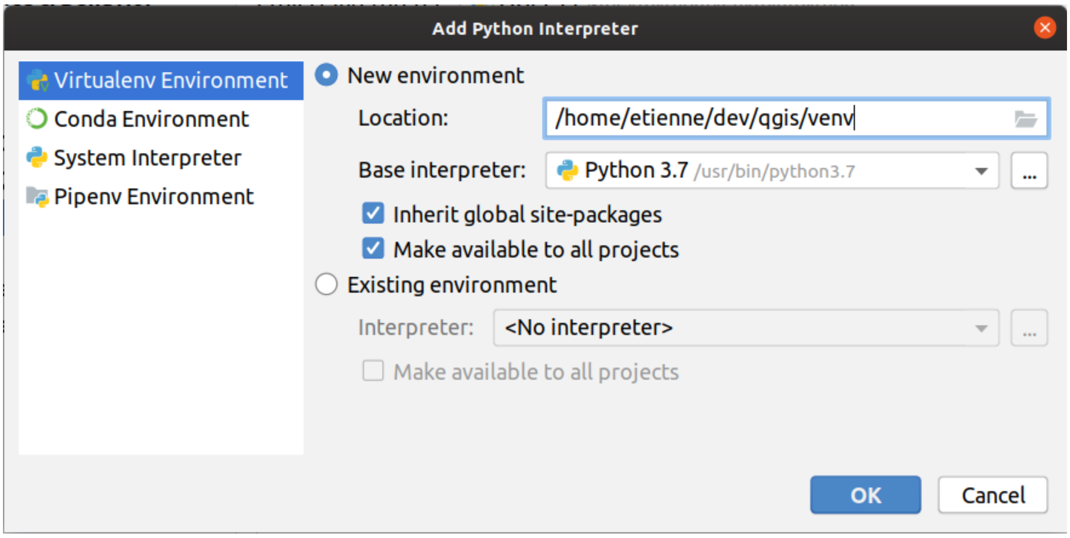
Note: Every time the QGIS API changes (e.g. if you're compiling QGIS master and the SIP file changed), you should go back to this page and simply click *Apply*. This will let Eclipse parse all the libraries again.

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

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*.

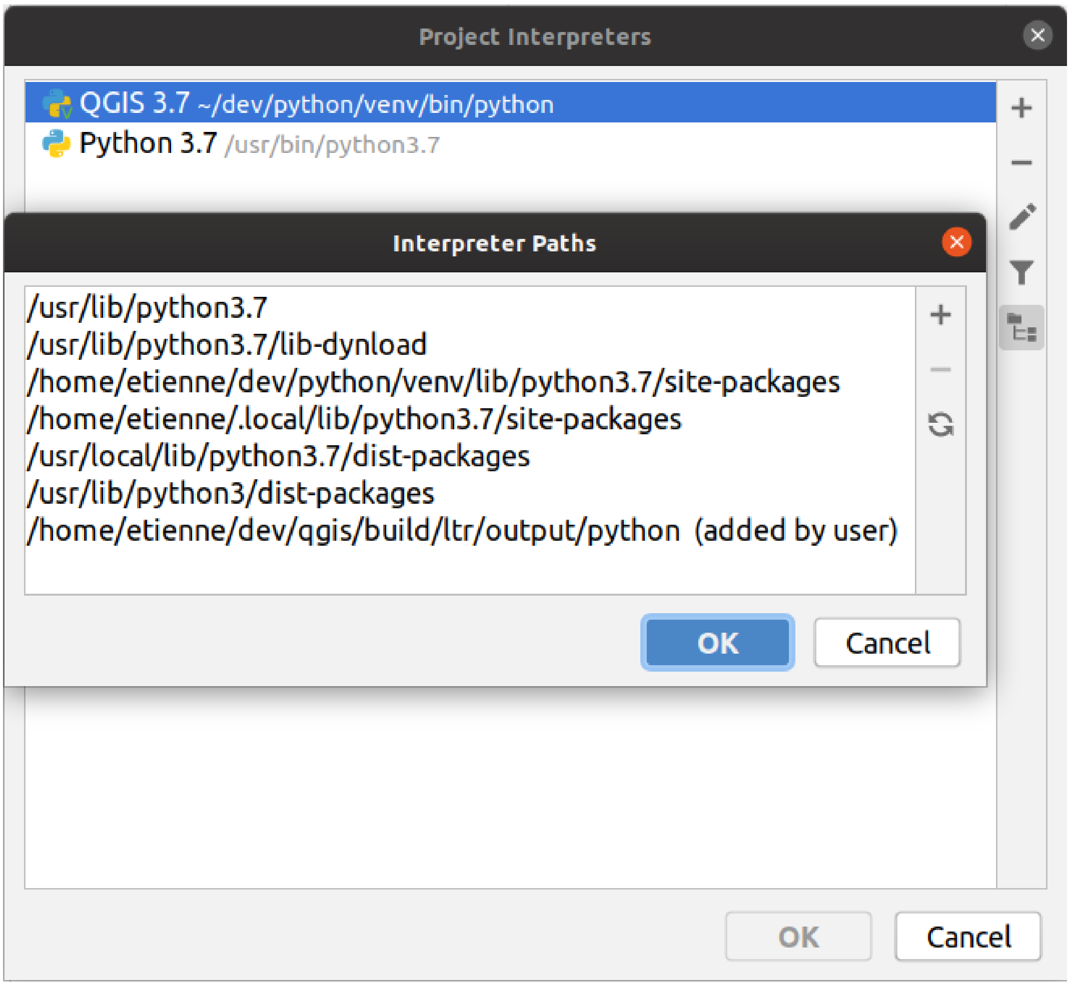


1. Click *OK*, come back on the small gear and click *Show all*.
2. In the new window, select your new interpreter `QGIS` and click the last icon in the vertical menu *Show paths for the selected interpreter*.
3. Finally, add the following absolute path to the list `~/dev/qgis/build/master/output/python`.

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.


Project Interpreters
QGIS 3.7 ~/dev/python/venv/bin/python
Python 3.7 /usr/bin/python3.7
Interpreter Paths
/usr/lib/python3.7
/usr/lib/python3.7/lib-dynload
/home/etienne/dev/python/venv/lib/python3.7/site-packages
/home/etienne/.local/lib/python3.7/site-packages
/usr/local/lib/python3.7/dist-packages
/usr/lib/python3/dist-packages
/home/etienne/dev/qgis/build/ltr/output/python (added by user)
OK
Cancel
OK
Cancel

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

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

2. Then run QGIS from the command line.

   On Linux do:

```
$ ./Qgis
```

   On macOS do:

```
$ /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

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*
- 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 it's 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
```

It is possible to create plugins in the Python programming language. In comparison with classical plugins written in C++ these should be easier to write, understand, maintain and distribute due to the dynamic nature of the Python language.

Python plugins are listed together with C++ plugins in QGIS plugin manager. They are searched for in `~/(UserProfile)/python/plugins` and these paths:

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

For definitions of ~ and `(UserProfile)` see core_and_external_plugins.

Note: By setting *QGIS_PLUGINPATH* to an existing directory path, you can add this path to the list of paths that are searched for plugins.

CHAPTER

SEVENTEEN

WRITING A PROCESSING 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:

```python
from qgis.core import QgsApplication
from processing_provider.provider import Provider

class YourPluginName():

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        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, **kwargs):
        """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, **kwargs):
        """The human friendly name of your plugin in Processing.

        This string should be as short as possible (e.g. "Lastools", not
        "Lastools version 1.0.1 64-bit") and localised.
        """
        return self.tr('Your plugin')

    def icon(self):
        """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,
)
```

CHAPTER

EIGHTEEN

NETWORK ANALYSIS LIBRARY

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)

The network analysis library was created by exporting basic functions from the RoadGraph core plugin and now you can use it's methods in plugins or directly from the Python console.

18.1 General information

Briefly, a typical use case can be described as:

1. create graph from geodata (usually polyline vector layer)
2. run graph analysis
3. use analysis results (for example, visualize them)

18.2 Building a graph

The first thing you need to do — is to prepare input data, that is to convert a vector layer into a graph. All further actions will use this graph, not the layer.

As a source we can use any polyline vector layer. Nodes of the polylines become graph vertexes, and segments of the polylines are graph edges. If several nodes have the same coordinates then they are the same graph vertex. So two lines that have a common node become connected to each other.

Additionally, during graph creation it is possible to "fix" ("tie") to the input vector layer any number of additional points. For each additional point a match will be found — the closest graph vertex or closest graph edge. In the latter case the edge will be split and a new vertex added.

Vector layer attributes and length of an edge can be used as the properties of an edge.

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 graphs 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 availabile. 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.

It's time to dive into the process.

First of all, to use this library we should import the analysis module

```
from qgis.analysis import *
```

Then some examples for creating a director

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.DirectionBoth)

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

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

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

And here is full list of what these parameters mean:

- `vectorLayer` — vector layer used to build the graph
- `directionFieldId` — index of the attribute table field, where information about roads direction is stored. If `-1`, then don't use this info at all. An integer.
- `directDirectionValue` — field value for roads with direct direction (moving from first line point to last one). A string.
- `reverseDirectionValue` — field value for roads with reverse direction (moving from last line point to first one). A string.
- `bothDirectionValue` — field value for bidirectional roads (for such roads we can move from first point to last and from last to first). A string.
- `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

It is necessary then to create a strategy for calculating edge properties

```
# 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)
```

And tell the director about this strategy

```
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 const:*True* (use OTF).
- `topologyTolerance` — topological tolerance. Default value is 0.
- `ellipsoidID` — ellipsoid to use. By default "WGS84".

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

Also we can define several points, which will be used in the analysis. For example

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

Now all is in place so we can build the graph and "tie" these points to it

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

Building the graph can take some time (which depends on the number of features in a layer and layer size). `tiedPoints` is a list with coordinates of "tied" points. When the build operation is finished we can get the graph and use it for the analysis

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

With the next code we can get the vertex indexes of our points

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

18.3 Graph analysis

Networks analysis is used to find answers to two questions: which vertexes are connected and how to find a shortest path. To solve these problems the network analysis library provides Dijkstra's algorithm.

Dijkstra's algorithm finds the shortest route from one of the vertexes of the graph to all the others and the values of the optimization parameters. The results can be represented as a shortest path tree.

The shortest path tree is a directed weighted graph (or more precisely a tree) with the following properties:

- only one vertex has no incoming edges — the root of the tree
- all other vertexes have only one incoming edge

- if vertex B is reachable from vertex A, then the path from A to B is the single available path and it is optimal (shortest) on this graph

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).

> **Warning:** 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)

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 Finding shortest paths

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
```

At this point we have the path, in the form of the inverted list of vertexes (vertexes are listed in reversed order from end point to start point) that will be visited during traveling by this path.

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 *
```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

```
builder = QgsGraphBuilder(vectorLayer.sourceCrs())

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

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

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

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

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

# Total cost
cost = costs[idxEnd]

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

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

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

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

18.3.2 Areas of availability

The area of availability for vertex A is the subset of graph vertexes that are accessible from vertex A and the cost of the paths from A to these vertexes are not greater that some value.

More clearly this can be shown with the following example: "There is a fire station. Which parts of city can a fire truck reach in 5 minutes? 10 minutes? 15 minutes?". Answers to these questions are fire station's areas of availability.

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.

A more difficult problem is to get the borders of the area of availability. The bottom border is the set of vertexes that are still accessible, and the top border is the set of vertexes that are not accessible. In fact this is simple: it is the availability border based on the edges of the shortest path tree for which the source vertex of the edge is accessible and the target vertex of the edge is not.

Here is an example

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

(continues on next page)

(continued from previous page)

```
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))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() + delta))

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

idStart = graph.findVertex(tStart)

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

upperBound = []
r = 1500.0
i = 0
tree.reverse()

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

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))
```

CHAPTER

NINETEEN

QGIS SERVER AND PYTHON

19.1 Introduction

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 OCG 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 OCG 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 standlone WSGI/HTTP service
- FILTERS → Enhance/Customize QGIS Server with filter plugins
- SERVICES → Add a new *SERVICE*
- OGC APIs → Add a new *OGC API*

Embeding and standalone applications require using the QGIS Server Python API directly from another Python script or application while 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)
- `QgsServerResponse` the response 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:

```
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 `QgsServerInterface`.

> **Warning:** 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:

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

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

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

# Create a response objects
response = QgsBufferServerResponse()

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

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

app.exitQgis()
```

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/master/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 instanciated by subclassing one of the classes below and by calling the corresponding method of `QgsServerInterface`:

Filter Type	Base Class	QgsServerInterface registration
I/O	`QgsServerFilter`	`registerFilter`
Access Control	`QgsAccessControlFilter`	`registerAccessControl`
Cache	`QgsServerCacheFilter`	`registerServerCache`

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 was needed (WFS GetFeature is one of them), in this case, `sendResponse` is called multiple times before the response is complete (and before `responseComplete` is called). The obvious consequence is that `sendResponse` is normally called once but might be exceptionally called multiple times 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 *Developing Python Plugins*, 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*)

```
server=True
```

Important: 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, 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

```
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

```
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"

```
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")
```

(continues on next page)

(continued from previous page)

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

The filters must be registered into the **serverIface** as in the following example:

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

```
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] HelloServerServer - loading filter ParamsFilter
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin HelloServer loaded!
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python plugins loaded
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair SERVICE // HELLO into the parameter map
```

(continues on next page)

(continued from previous page)

```
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 here below).

Tip: 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)`

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:

```
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" % 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:

```
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)

```

(continues on next page)

(continued from previous page)

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

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

    def allowToEdit(self, layer, feature):
        """ Are we authorise 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(self.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.:

```
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)

```
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:

```
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:

```
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:

```
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:

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

class CustomServiceService(QgsService):

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

    def name(self):
```

(continues on next page)

(continued from previous page)

```
        return "CUSTOM"

    def version(self):
        return "1.0.0"

    def allowMethod(method):
        return True

    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 `QgsServerOgcApiHandler` subclasses that are registered to an instance of `QgsServerOgcApi` (or it's 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`:

```
import json
import os

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

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


class CustomApiHandler(QgsServerOgcApiHandler):

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

```

(continues on next page)

(continued from previous page)

```python
    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)

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

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


class CustomApi():

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

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

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

from qgis.core import (
```

(continues on next page)

(continued from previous page)

```
    QgsProject,
    QgsLayerTreeModel,
)

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

CHAPTER

TWENTY

CHEAT SHEET FOR PYQGIS

20.1 User Interface

Change Look & Feel

```
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

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon("/path/to/logo/file.png")
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("My QGIS")
```

20.2 Settings

Get QgsSettings list

```
from qgis.core import QgsSettings

qs = QgsSettings()

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

20.3 Toolbars

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.PyQt.QtCore 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 Layers

Add vector layer

```
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

```
layer = iface.activeLayer()
```

List all layers

```
from qgis.core import QgsProject

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

Obtain layers name

```
from qgis.core import QgsVectorLayer
layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer)

layers_names = []
for layer in QgsProject.instance().mapLayers().values():
    layers_names.append(layer.name())

print("layers TOC = {}".format(layers_names))
```

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

Otherwise

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

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

Find layer by name

```
from qgis.core import QgsProject

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

```
layer name you like
```

Set active layer

```
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)
```

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

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
for f in layer.getFeatures():
    geom = f.geometry()
    print ('%f, %f' % (geom.asPoint().y(), geom.asPoint().x()))
```

```
10.000000, 10.000000
```

Move geometry

```
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

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

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

See the CRS

```
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

```
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

```
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))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])
```

Load all vector layers from GeoPackage

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

for subLayer in subLayers:
    name = subLayer.split('!!::!!')[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)

```
from qgis.core import QgsRasterLayer, QgsProject

def loadXYZ(url, name):
    rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
    QgsProject.instance().addMapLayer(rasterLyr)

urlWithParams = 'type=xyz&url=https://a.tile.openstreetmap.org/%7Bz%7D/%7Bx%7D/%7By
↪%7D.png&zmax=19&zmin=0&crs=EPSG3857'
loadXYZ(urlWithParams, 'OpenStreetMap')
```

Remove all layers

```
QgsProject.instance().removeAllMapLayers()
```

Remove all

```
QgsProject.instance().clear()
```

20.7 Table of contents

Access checked layers

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

Remove contextual menu

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

20.8 Advanced TOC

Root node

```
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

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer, QgsLayerTree

child0 = root.children()[0]
print (child0.name())
print (type(child0))
print (isinstance(child0, QgsLayerTreeLayer))
print (isinstance(child0.parent(), QgsLayerTree))
```

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

Find groups and nodes

```
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
```

Find group by name

```
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

```
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

```
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

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

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

(continues on next page)

(continued from previous page)

```

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

parent.removeChildNode(myLayer)
```

Move loaded layer to a specific group

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

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

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]
node = root.findLayer( layer.id())

index = model.node2index( node )
ltv.setRowHidden( index.row(), index.parent(), True )
node.setCustomProperty( 'nodeHidden', 'true')
ltv.setCurrentIndex(model.node2index(root))
```

Node signals

```
def onWillAddChildren(node, indexFrom, indexTo):
    print ("WILL ADD", node, indexFrom, indexTo)

def onAddedChildren(node, indexFrom, indexTo):
```

(continues on next page)

(continued from previous page)

```
    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)
view = QgsLayerTreeView()
view.setModel(model)
view.show()
```

Move node

```
cloned_group1 = node_group.clone()
root.insertChildNode(0, cloned_group1)
root.removeChildNode(node_group)
```

Rename node

```
cloned_group1.setName("Group X")
node_layer1.setName("Layer X")
```

20.9 Processing algorithms

Get algorithms list

```
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

Random selection

```
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.

```
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?

```
len(QgsApplication.processingRegistry().algorithms())
```

How many providers are there?

```
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

```
from qgis.PyQt.Qt import QTextDocument
from qgis.PyQt.QtGui import QFont

mQFont = "Sans Serif"
mQFontsize = 9
mLabelQString = "© QGIS 2019"
mMarginHorizontal = 0
mMarginVertical = 0
mLabelQColor = "#FF0000"

INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
case = 2

def add_copyright(p, text, xOffset, yOffset):
    p.translate( xOffset , yOffset  )
    text.drawContents(p)
    p.setWorldTransform( p.worldTransform() )

def _on_render_complete(p):
    deviceHeight = p.device().height() # Get paint device height on which this painter is currently painting
    deviceWidth  = p.device().width() # Get paint device width on which this painter is currently painting
    # Create new container for structured rich text
    text = QTextDocument()
    font = QFont()
    font.setFamily(mQFont)
    font.setPointSize(int(mQFontsize))
    text.setDefaultFont(font)
    style = "<style type=\"text/css\"> p {color: " + mLabelQColor + "}</style>"
    text.setHtml( style + "<p>" + mLabelQString + "</p>" )
    # Text Size
    size = text.size()
```

(continues on next page)

(continued from previous page)

```

    # RenderMillimeters
    pixelsInchX  = p.device().logicalDpiX()
    pixelsInchY  = p.device().logicalDpiY()
    xOffset  = pixelsInchX  * INCHES_TO_MM * int(mMarginHorizontal)
    yOffset  = pixelsInchY  * INCHES_TO_MM * int(mMarginVertical)

    # Calculate positions
    if case == 0:
        # Top Left
        add_copyright(p, text, xOffset, yOffset)

    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

```
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