PyQGIS developer cookbook

Release 3.4

QGIS Project

out 25, 2019

Contents

CHAPTER 1

Introdução

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 custom applications based on the QGIS API

Python bindings are also available for QGIS Server, including Python plugins (see *Complementos de servidores Python do QGIS*) 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:
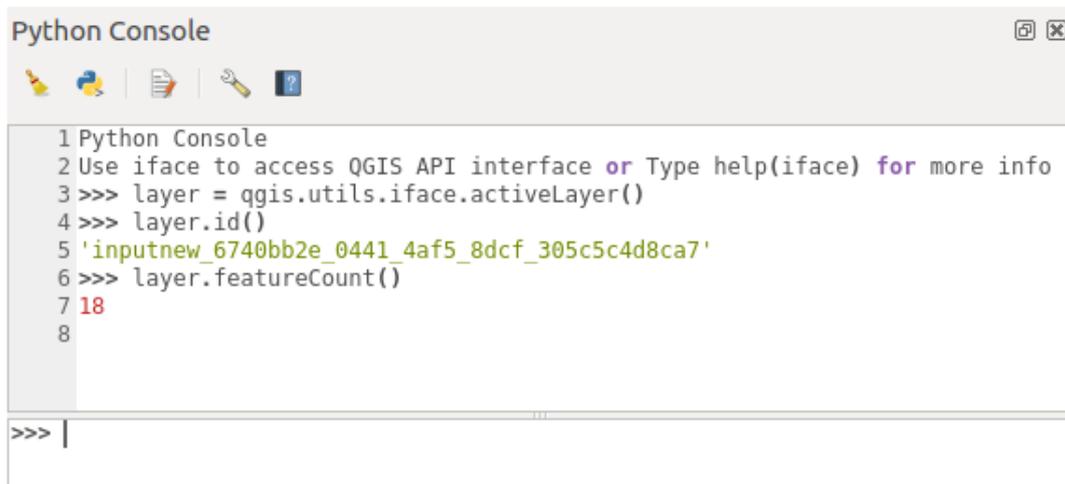
Figure 1.1: Terminal Python QGIS

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 Complementos Python

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.

Nota: Python plugins are also available for QGIS server. See *Complementos de servidores Python do QGIS* for further details.

1.3 Running Python code when QGIS starts

Existem dois métodos distintos de executar o código em Phyton toda vez que o QGIS é iniciado.

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.

Nota: 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 Aplicações Python

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.

Nota: 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
# 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 Executando aplicativos personalizados

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.

Carregando projetos

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

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

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

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

```python
# Save the project to the same
project.write()
# ... or to a new file
project.write('/home/user/projects/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.

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

```python
bridge = QgsLayerTreeMapCanvasBridge( \
         QgsProject.instance().layerTreeRoot(), canvas)
```

```
# Now you can safely load your project and see it in the canvas
project.read('/home/user/projects/my_other_qgis_project.qgs')
```

Carregando Camadas

The code snippets on this page needs the following imports:

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

- *Camadas Vetoriais*
- *Camadas Matriciais*
- *QgsProject instance*

Vamos abrir algumas camadas com dados. QGIS reconhece camadas vetoriais e matriciais. Adicionalmente, camadas personalizadas estão disponíveis, mas não discutiremos este tipo de camadas aqui.

3.1 Camadas Vetoriais

To create a vector layer instance, specify layer's data source identifier, name for the layer and provider's name:

```python
# get the path to the shapefile e.g. /home/project/data/ports.shp
path_to_ports_layer = os.path.join(QgsProject.instance().homePath(), "data", "ports
→", "ports.shp")

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

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

O identificador da fonte de dados é uma string e é especifico para cada provedor de dados vetoriais. O nome da camada é usado no painel de lista de camadas. É importante verificar se a camada foi carregada com sucesso. Se não for, uma instância de camada inválida é retornada.

For a geopackage vector layer:

```
# get the path to a geopackage  e.g. /home/project/data/data.gpkg
path_to_gpkg = os.path.join(QgsProject.instance().homePath(), "data", "data.gpkg")
# append the layername part
gpkg_places_layer = path_to_gpkg + "|layername=places"
# e.g. gpkg_places_layer = "/home/project/data/data.gpkg|layername=places"
vlayer = QgsVectorLayer(gpkg_places_layer, "Places layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")
```

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

```
vlayer = iface.addVectorLayer(path_to_ports_layer, "Ports 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.

A lista a seguir mostra como acessar várias fontes de dados usando provedores de dados vetoriais:

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

    - for Shapefile:

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

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

        ```
        uri = "/path/to/dxffile/file.dxf|layername=entities|geometrytype=Point"
        vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
        ```

- 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")

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

    Nota: 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 = "/some/path/file.csv?delimiter={}&xField={}&yField={}".format(";", "x",
    ↪"y")
    vlayer = QgsVectorLayer(uri, "layer name you like", "delimitedtext")
    ```

    Nota: 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 = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer name you like", "gpx")
```

- 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')
```

- 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" )
```

- WFS connection:. the connection is defined with a URI and using the `WFS` provider:

```
uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=union&
→version=1.0.0&request=GetFeature&service=WFS",
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")
```

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

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

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

```
# vlayer is a vector layer, uri is a QgsDataSourceUri instance
vlayer.setDataSource(uri.uri(), "layer name you like", "postgres")
```

3.2 Camadas Matriciais

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

```python
# get the path to a tif file  e.g. /home/project/data/srtm.tif
path_to_tif = os.path.join(QgsProject.instance().homePath(), "data", "srtm.tif")
rlayer = QgsRasterLayer(path_to_tif, "SRTM layer name")
if not rlayer.isValid():
    print("Layer failed to load!")
```

To load a raster from a geopackage:

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

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


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

Similarly to vector layers, raster layers can be loaded using the addRasterLayer function of the `QgisInterface` object:

```python
iface.addRasterLayer("/path/to/raster/file.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.

Raster layers can also be created from a WCS service:

```python
layer_name = 'modis'
uri = QgsDataSourceUri()
uri.setParam('url', 'http://demo.mapserver.org/cgi-bin/wcs')
uri.setParam("identifier", layer_name)
rlayer = QgsRasterLayer(str(uri.encodedUri()), '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 = 'url=http://irs.gis-lab.info/?layers=landsat&styles=&format=image/
↪jpeg&crs=EPSG:4326'
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()
```

Usando Camadas Raster

Aviso: *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.*

- *Detalhes da Camada*
- *Renderer*
    - *Rasters única banda*
    - *Rasters Multi Banda*
- *Query Values*

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

```python
from qgis.core import (
  QgsRasterLayer,
  QgsColorRampShader,
  QgsSingleBandPseudoColorRenderer
)
```

4.1 Detalhes da Camada

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

The following code assumes rlayer is a QgsRasterLayer object.

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

```
(919, 619)
# get the extent of the layer as QgsRectangle
rlayer.extent()
<QgsRectangle: 20.06856808199999875 -34.27001076999999896, 20.83945284300000012 -
→33.75077500700000144>
# get the extent of the layer as Strings
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
rlayer.rasterType()
0
# get the total band count of the raster
rlayer.bandCount()
1
# get all the available metadata as a QgsLayerMetadata object
rlayer.metadata()
'<qgis._core.QgsLayerMetadata object at 0x13711d558>'
```

4.2 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:

```
rlayer.renderer()
<qgis._core.QgsSingleBandGrayRenderer object at 0x7f471c1da8a0>
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.

4.2.1 Rasters única banda

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()
```

4.2.2 Rasters Multi Banda

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

```
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()
```

4.3 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())
```

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}`

Usando Camadas Vetor

Esta seção lista várias operações que podem ser realizadas com camadas vetoriais.

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

5.1 Retrieving information about attributes

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

```python
# "layer" is a QgsVectorLayer instance
for field in layer.fields():
    print(field.name(), field.typeName())
```

5.2 Interagindo sobre camada vetor

Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. The `layer` variable is assumed to have a `QgsVectorLayer` object.

```python
layer = 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:
            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)
```

5.3 Selecionando características

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
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()
```

5.3.1 Acessando atributos

Attributes can be referred to by their name:

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

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

```
print(feature[0])
```

5.3.2 Iteração sobre os feições selecionadas

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

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

5.3.3 Iterando sobre um subconjunto de feições

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
```

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):
    # loop through only 2 features
```

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 *Expressões, filtragem e cálculo dos valores* 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])
```

5.4 Modificando Camadas Vetoriais

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')
```

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,
# 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*.

Nota: 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()
```

5.4.1 Adicionar feições

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])
```

5.4.2 Excluir feições

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

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

5.4.3 Modificar Feições

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

Dica: 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.).

5.4.4 Modificando Camadas Vetoriais com um 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 commited, all changes from the editing buffer are saved to data provider.

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

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

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

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

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

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

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

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

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

Here is how you can use the undo functionality:

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

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

if problem_occurred:
  layer.destroyEditCommand()
  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:

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

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

5.4.5 Adicionando e Removendo Campos

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

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

if caps & QgsVectorDataProvider.AddAttributes:
```

```
    res = layer.dataProvider().addAttributes(
        [QgsField("mytext", QVariant.String),
         QgsField("myint", QVariant.Int)])

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

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()
```

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

Using `with edit(layer):` the changes will be commited automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollBack()` all the changes. See *Modificando Camadas Vetoriais com um Buffer*.

5.5 Utilizando Índices Espaciais

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.insertFeature(feat)
```

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

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

- uma vez que o índice espacial é preenchido com alguns valores, você pode fazer algumas consultas

```
# 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))
```

5.6 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`...).

5.6.1 From an instance of `QgsVectorFileWriter`

```
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormat(layer,
                                                "/path/to/folder/my_data",
                                                "")
if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
```

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
error = QgsVectorFileWriter.writeAsVectorFormat(layer,
                                                "/path/to/folder/my_esridata",
                                                "UTF-8",
                                                driverName="ESRI Shapefile")
if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
```

The third (mandatory) parameter specifies output text encoding. Only some drivers need this for correct operation - Shapefile is one of them (other drivers will ignore this parameter). Specifying the correct encoding is important if you are using international (non US-ASCII) characters.

```
# Write to an ESRI GDB file
opts = QgsVectorFileWriter.SaveVectorOptions()
opts.driverName = "FileGDB"
# if no geometry
opts.overrideGeometryType = QgsWkbTypes.NullGeometry
opts.actionOnExistingFile = QgsVectorFileWriter.ActionOnExistingFile.
→CreateOrOverwriteLayer
opts.layerName = 'my_new_layer_name'
error = QgsVectorFileWriter.writeAsVectorFormat(layer=vlayer,
                                                fileName=gdb_path,
                                                options=opts)
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
      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 is a QgsVectorFileWriter.SaveVectorOptions as above
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.

5.6.2 Directly from features

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

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

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

writer = QgsVectorFileWriter("my_shapes.shp", "UTF-8", fields, QgsWkbTypes.Point,
→driverName="ESRI Shapefile")

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

5.6.3 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"`, or `"MultiPolygon"`.

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

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

index=yes Especifica que o provedor irá usar o index espacial

field=name:type(tamanho,precisão) 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.

O exemplo seguinte de URL incorpora todas estas opções

```
"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
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())
```

5.7 Aparencia (Simbologia) de Camadas de Vetor

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())
```

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

Tipo	Classes	Descrição
singleSymbol	QgsSingleSymbolRenderer	Renderiza todas as características com o mesmo símbolo
catego-rizedSymbol	QgsCategorizedSymbolRenderer	Renderiza características usando um símbolo diferente para cada categoria
graduat-edSymbol	QgsGraduatedSymbolRenderer	Renderiza caracter´sticas usando diferents símbolos para cada limite de valores

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())
# Print:
['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

```
print(renderer.dump())
```

5.7.1 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:

- circulo
- quadrado
- cross
- *retângulo'*
- diamante
- pentágono
- triângulo
- triângulo_equilateral
- estrela
- regular_star
- flecha
- 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())
# Prints
{'angle': '0',
'color': '0,128,0,255',
'horizontal_anchor_point': '1',
'joinstyle': 'bevel',
'name': 'circle',
'offset': '0,0',
'offset_map_unit_scale': '0,0',
```

```
'offset_unit': 'MM',
'outline_color': '0,0,0,255',
'outline_style': 'solid',
'outline_width': '0',
'outline_width_map_unit_scale': '0,0',
'outline_width_unit': 'MM',
'scale_method': 'area',
'size': '2',
'size_map_unit_scale': '0,0',
'size_unit': 'MM',
'vertical_anchor_point': '1'}
```

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

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

5.7.2 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

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

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

The renderer usually stores also original symbol and color ramp which were used for the classification: `sourceColorRamp()` and `sourceSymbol()` methods.

5.7.3 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

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

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)

```python
from qgis.PyQt import QtGui

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = 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)
myRenderer.setMode(QgsGraduatedSymbolRenderer.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
QgsProject.instance().addMapLayer(myVectorLayer)
```

5.7.4 Trabalhando com Símbolos

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

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

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

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

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

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.

Por padrão, tamanho e largura são em milimetros e ângulos em graus.

Trabalhando com Camadas de Símbolos

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)
```

Output:

```
EllipseMarker
FilledMarker
FontMarker
GeometryGenerator
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
```

```python
    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

```python
from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

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

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

    def symbolLayer(self):
        return self.layer

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

This widget can be embedded into the symbol properties dialog. When the symbol layer type is selected in symbol properties dialog, it creates an instance of the symbol layer and an instance of the symbol layer widget. Then it calls the `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.

Você terá que criar metadados para a camada de símbolos

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

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

  def __init__(self):
    QgsSymbolLayerAbstractMetadata.__init__(self, "FooMarker", QgsSymbol.Marker)

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

      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.

O último passo para adicionar este símbolo de camada para o registro — e estamos prontos.

5.7.5 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

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


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

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

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

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

```
      s.stopRender(context)

  def usedAttributes(self):
    return []

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

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

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

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

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

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

```
    self.r.syms[0].setColor(color)
    self.btn1.setColor(self.r.syms[0].color())

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

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 Random-Renderer example

```
from qgis.core import QgsRendererAbstractMetadata,QgsRendererRegistry,
→QgsApplication


class RandomRendererMetadata(QgsRendererAbstractMetadata):
  def __init__(self):
    QgsRendererAbstractMetadata.__init__(self, "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).

5.8 Outros Tópicos

TODO:

- creating/modifying symbols

- working with style (`QgsStyle`)

- working with color ramps (`QgsColorRamp`)

- exploring symbol layer and renderer registries

CHAPTER 6

Manipulação Geométrica

- *Construção de Geométria*
- *Acesso a Geometria*
- *Operações e Predicados Geométricos*

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

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

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

Às vezes, uma geometria é realmente uma coleção dex simples geometrias (single-part). Tal geometria é chamada de geometria de várias partes. Se ele contém apenas um tipo de simples geometria, podemos chamar de multi-ponto, multi-cadeia linear ou multi-polígono. Por exemplo, um país que consiste de múltiplas ilhas pode ser representado como um sistema multi-polígono.

As coordenadas de geometrias podem estar em qualquer sistema de referência de coordenadas (SRC). Ao buscar feições a partir de uma camada, geometrias associadas terão coordenadas no SRC da camada.

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

6.1 Construção de Geométria

PyQGIS provides several options for creating a geometry:

- a partir das coordenadas

```
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.

Geometrias multi-parte passam para um nível maior: multi-ponto é uma lista de pontos, multi-cadeia linear é uma lista de cadeias lineares e multi-polígono é uma lista de polígonos.

- a partir de textos conhecidos (WKT)

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

- a partir de binários conhecidos (WKB)

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

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

6.2 Acesso a Geometria

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())
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)>]]
```

Nota: 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()`.

6.3 Operações e Predicados Geométricos

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

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

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

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

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

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

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

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

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

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

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

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

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

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))
```

Você pode encontrar muitos exemplo de algoritmos que estão incluídos no QGIS e usar esses métodos para analisar e transformar dados vetoriais. Aqui estão alguns links para o código de alguns deles.

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

Suporte a projeções

- *Sistemas de Referencia de Coordenadas*
- *CRS Transformation*

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

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

7.1 Sistemas de Referencia de Coordenadas

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

- especificar CRS pela sua identificação

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

  O QGIS usa três diferentes ID para cada sistema de referencia:

  - `InternalCrsId` — ID used in the internal QGIS database.
  - `PostgisCrsId` — ID used in PostGIS databases.
  - `EpsgCrsId` — ID assigned by the EPSG organization.

  If not specified otherwise with the second parameter, PostGIS SRID is used by default.

- especificar CRS pelo seu texto conhecido (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 Proj4 string to initialize the projection.

```
crs = QgsCoordinateReferenceSystem()
crs.createFromProj4("+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(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("Proj4 String:", crs.toProj4())
# 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
Proj4 String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

7.2 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(4326)    # WGS 84
crsDest = QgsCoordinateReferenceSystem(32633)  # WGS 84 / UTM zone 33N
xform = QgsCoordinateTransform(crsSrc, crsDest, QgsProject.instance())
```

```
# 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)>
```

Using the Map Canvas

- *Incorporar o Mapa da Tela*
- *Bandas raster e fazedor de vértices*
- *Usando Ferramentas de Mapas na Tela*
- *Desenhar ferramenta de mapa personalizada*
- *Desenhar itens da tela do mapa*

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.

Para resumir, a arquitetura do mapa na tela são constituídas por três conceitos:

- tela do mapa — para visualização do mapa
- map canvas items — additional items that can be displayed on the map canvas
- map tools — for interaction with the map canvas

8.1 Incorporar o Mapa da Tela

Mapa natela é um widget, como qualquer outro widget Qt, então usá-lo é tão simples como criar e mostrá-lo

```
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.

Por padrão, o mapa na tela tem fundo preto e não usa anti-aliasing. Para definir o fundo branco e permitir anti-aliasing para renderização suave

```
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 canvas

```
path_to_ports_layer = os.path.join(QgsProject.instance().homePath(), "data", "ports
→", "ports.shp")

vlayer = QgsVectorLayer(path_to_ports_layer, "Ports 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])
```

Depois de executar esses comandos, a tela deve mostrar a camada que você carregou.

8.2 Bandas raster e fazedor de vértices

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.

Para mostrar as polilinhas

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

Para mostrar o poligono

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

Note-se que aponta para polígono não é uma lista simples: na verdade, é uma lista de anéis contendo anéis lineares do polígono: primeiro anel é a borda externa, ainda mais (opcional) anéis correspondem aos buracos no polígono.

As Bandas Raster permitem alguma personalização, ou seja, para mudar sua cor e linha de largura

```
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)
```

(em C ++ é possível simplesmente apagar o item, no entanto, em Python `` del r`` seria apenas suprimir a referência eo objeto continuará a existir, uma vez que é de propriedade da tela)

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.

8.3 Usando Ferramentas de Mapas na Tela

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

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

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

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

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

        self.setCentralWidget(self.canvas)

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

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

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

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

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

        self.pan()

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

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

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

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

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

8.4 Desenhar ferramenta de mapa personalizada

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.

Aqui está um exemplo de uma ferramenta de mapa que permite definir uma medida retangular, clicando e arrastando na tela. Quando o retângulo é definido, ele imprime coordena seu limite no console. Ele utiliza os elementos

de banda de borracha descritos antes para mostrar o retângulo selecionado, uma vez que está a ser definida.

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

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

8.5 Desenhar itens da tela do mapa

TODO: Como criar itens do mapa na tela

```python
import sys
from qgis.core import QgsApplication
from qgis.gui import QgsMapCanvas

def init():
  a = QgsApplication(sys.argv, True)
  QgsApplication.setPrefixPath('/home/martin/qgis/inst', True)
  QgsApplication.initQgis()
  return a

def show_canvas(app):
  canvas = QgsMapCanvas()
  canvas.show()
  app.exec_()
app = init()
show_canvas(app)
```

Renderização em impressão de mapas

The code snippets on this page needs the following imports:

```python
import os
```

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.

9.1 Renderização simples

The rendering is done creating a `QgsMapSettings` object to define the rendering options, and then constructing a `QgsMapRendererJob` with those options. 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")

# e.g. vlayer = iface.activeLayer()
vlayer = QgsProject.instance().mapLayersByName("countries")[0]
options = QgsMapSettings()
options.setLayers([vlayer])
options.setBackgroundColor(QColor(255, 255, 255))
options.setOutputSize(QSize(800, 600))
options.setExtent(vlayer.extent())

render = QgsMapRendererParallelJob(options)

def finished():
```

```
    img = render.renderedImage()
    # save the image; e.g. img.save("/Users/myuser/render.png","png")
    img.save(image_location, "png")
    print("saved")

render.finished.connect(finished)

render.start()
```

9.2 Renderizando camadas com CRS diferente

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

```
settings.setLayers(layers)
render.setDestinationCrs(layers[0].crs())
```

9.3 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:

```
p = QgsProject()
layout = QgsLayout(p)
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 — este item diz as bibliotecas onde colocar o próprio mapa. Aqui criamos um mapa e esticamos sobre o tamanho do papel inteiro

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

- label — permite exibir rótulos. É possível modificar a sua fonte, cor, alinhamento e margem

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

- legenda

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

- barra de escala

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

- Seta

- Imagem

- 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)
```

- Tabela

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.

9.3.1 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.

9.3.2 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 needs 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
)
```

Expressões, filtragem e cálculo dos valores

O QGIS possui alguns recursos para análise de expressões semelhantes à SQL. Apenas um pequeno subconjunto da sintaxe SQL é suportado. As expressões podem ser avaliadas tanto como predicados booleanos (retornando Verdadeiro ou Falso) ou como funções (retornando um valor escalar) . Veja vector_expressions no Manual do Usuário para uma lista completa das funções disponíveis.

Três tipos básicos são suportados:

- número — ambos os números inteiros e números decimais, por exemplo, `123`, `3.14`
- texto — eles devem estar entre aspas simples: *'Olá world''*
- coluna referência — ao avaliar, a referência é substituída com o valor real do campo. Os nomes não alteram.

As seguintes operações estão disponíveis:

- operadores aritméticos: `+`, `-`, `*`, `/`, `^`
- parênteses: para fazer cumprir a precedência do operador: `(1 + 1) * 3`
- Sinal mais e menos: `-12`, `+5`
- funções matemáticas: `sqrt`, `sen`, `cos`, `tan`, `asen`, `acos`, `atan`
- Funções de conversão: "to_int", "to_real", "to_string", "to_date"
- funções geométricas: `$area`, `$length`
- funções de manipulação de geometria: "$x", "$y", "$geometry", "num_geometries", "centroid"

E os seguintes predicados são suportados:

- comparação: `=`, `!=`, `>`, `>=`, `<`, `<=`

- correspondência padrão: LIKE (usando % e _), ~ (expressões regulares)

- predicados lógicos: AND, OR, NOT

- verificação de valor nulo: IS NULL, IS NOT NULL

Exemplos de predicados:

- 1 + 2 = 3

- sen(ângulo) > 0

- 'Hello' LIKE 'He%'

- (x > 10 AND y > 10) OR z = 0

Exemplos de expressões escalares:

- 2 ^ 10

- sqrt(val)

- $length + 1

10.1 Expressões de Análise

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')
```

10.2 Expressões de Avaliação

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.

10.2.1 Expressões Básicas

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

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

10.2.2 Expressões com características

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:

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

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

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

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

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'])
```

10.2.3 Filtering a layer with expressions

O exemplo seguinte pode ser usado para filtrar uma camada e devolver qualquer característica que corresponda a um predicado.

```
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)
```

10.3 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 needs the following imports if you're outside the pyqgis console:

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

Leitura e Armazenamento de Configurações

Aviso: *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.*

Muitas vezes é útil para o plugin salvar algumas variáveis para que o utilizador não necessite introduzir ou selecionar outra vez numa próxima vez que o plugin for acionado.

Estas variáveis podem ser salvas e recuperadas com a ajuda do Qt e QGIS API. Para cada variável, você deve pegar a chave que será usada para acessar a variável — para cor favorita do usuário use a chave "favourite_color" ou alguma outra palavra com significado. É recomendado dar alguma estrutura para criação do nome das chaves.

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. The `setValue()` and `value()` methods from this class provide

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

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

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

  The second parameter of the `value()` method is optional and specifies the default value that is returned if there is no previous value set for the passed setting name.

- **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)
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 :meth:`writeEntry() <qgis.core.QgsProject.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")
```

Comunicação com o usuário

- *Showing messages. The QgsMessageBar class*
- *Mostrando progresso*
- *Carregando*

Esta seção mostra alguns métodos e elementos que devem ser usados para se comunicar com o usuário, a fim de manter a consistência na interface do usuário.

12.1 Showing messages. The QgsMessageBar class

Usando caixas de mensagem pode ser uma má idéia, do ponto de vista da experiência do usuário. Para mostrar uma pequena linha de informação ou uma mensagem de aviso/erro, a barra de mensagens QGIS é geralmente uma opção melhor.

Usando a referência ao objeto de interface QGIS, você pode mostrar uma mensagem na barra de mensagem com o seguinte código

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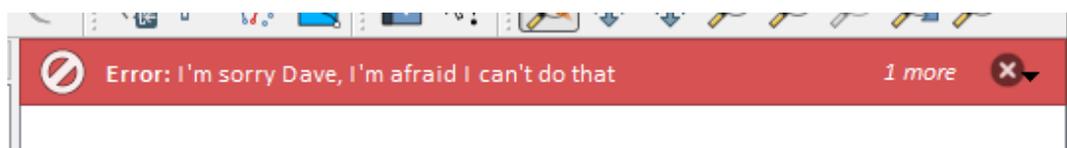

Figure 12.1: Barra de mensagem do QGIS

Você pode definir uma duração de mostrá-lo por um tempo limitado

```
iface.messageBar().pushMessage("Ooops", "The plugin is not working as it should",␣
↪level=Qgis.Critical, duration=3)
```
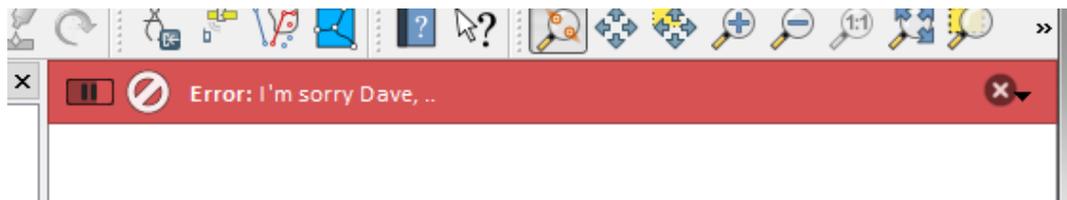
Figure 12.2: Barra de Mensagem QGIS como temporizar

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

0. Info

1. Warning

2. Critical

3. Success
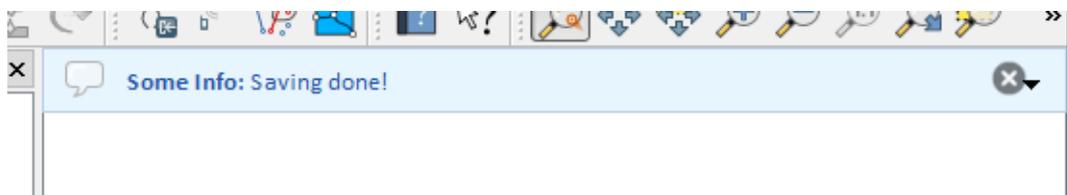
Figure 12.3: Barra de mensagens QGIS (informação)

Widgets podem ser adicionados à barra de mensagens, como por exemplo, um botão para mostrar mais informações

```python
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)
```
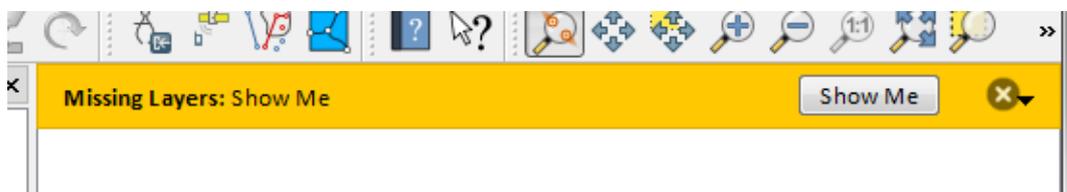

Figure 12.4: Barra de mensagens QGIS com um botão

Você ainda pode usar uma barra de mensagens em sua própria caixa de diálogo para que você não tenha de mostrar uma caixa de mensagem, ou se ela não faz sentido para mostrá-la na janela principal QGIS

```python
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0, 0, 0, 0)
```

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

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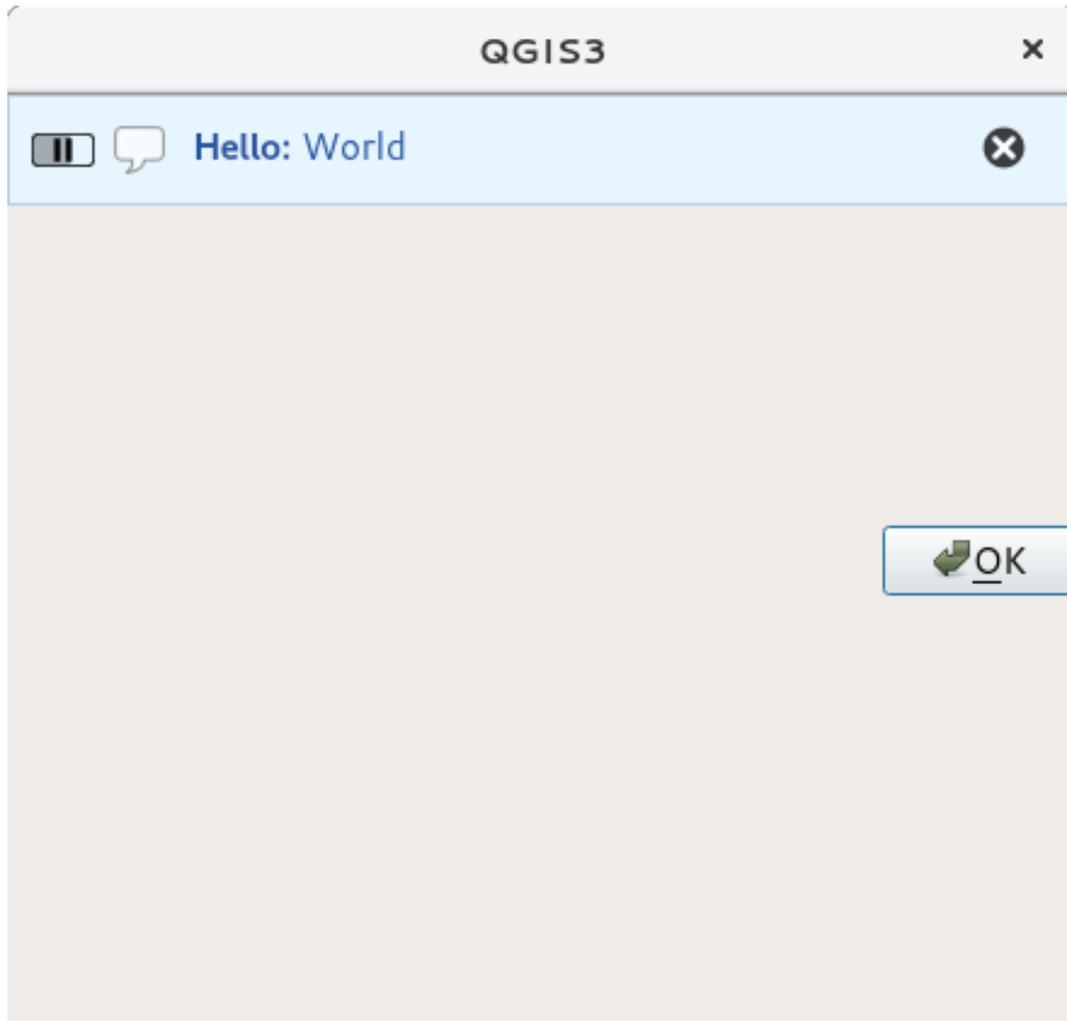

Figure 12.5: Barra de mensagens QGIS em diálogo personalizado

12.2 Mostrando progresso

As barras de progresso também pode ser colocado na barra de mensagem QGIS, uma vez que, como vimos, ele aceita widgets. Aqui está um exemplo que você pode tentar no console.

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

```
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)

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

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

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

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

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()
```

12.3 Carregando

Você pode usar o sistema de registro de QGIS para registrar todas as informações que você deseja salvar sobre a execução de seu código.

```
# 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)
```

Aviso: Use of the Python `print` statement is unsafe to do in any code which may be multithreaded. This includes **expression functions**, **renderers**, **symbol layers** and **Processing algorithms** (amongst others). In these cases you should always use thread safe classes (`QgsLogger` or `QgsMessageLog`) instead.

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

---

Nota:

- `QgsLogger` is for messages for debugging / developers (i.e. you suspect they are triggered by some broken code)

- `QgsMessageLog` is for messages to investigate issues by sysadmins (e.g. to help a sysadmin to fix configurations)

Authentication infrastructure

Aviso: *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.*

13.1 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.

Most of the following snippets are derived from the code of Geoserver Explorer plugin and its tests. This is the first plugin that used Authentication infrastructure. The plugin code and its tests can be found at this link. Other good code reference can be read from the authentication infrastructure tests code.

13.2 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.

13.3 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 master password or by transparently using it to access crypted stored info.

13.3.1 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 = QgsAuthManager.instance()
# check if QgsAuthManager has been already 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.authenticationDbPath():
    # 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
    # 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" )
```

13.3.2 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 QGIS API or GUI.

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

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

Available Authentication methods

*Authentication Method*s are loaded dynamically during authentication manager init. The list of Authentication method can vary with QGIS evolution, but the original list of available methods is:

1. `Basic` User and password authentication
2. `Identity-Cert` Identity certificate authentication
3. `PKI-Paths` PKI paths authentication
4. `PKI-PKCS#12` PKI PKCS#12 authentication

The above strings are that identify authentication methods in the QGIS authentication system. In Development section is described how to create a new c++ *Authentication Method*.

Populate Authorities

```
authMgr = QgsAuthManager.instance()
# 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()
```

> **Aviso:** Due to QT4/OpenSSL interface limitation, updated cached CA are exposed to OpenSsl only almost a minute later. Hope this will be solved in QT5 authentication infrastructure.

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
boundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                     "/path/to/alice-key_w-pass.pem",
                                     "unlock_pwd",
                                     "list_of_CAs_to_bundle" )
assert boundle is not None
assert boundle.isValid()
```

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

13.3.3 Remove entry from authdb

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

```
authMgr = QgsAuthManager.instance()
authMgr.removeAuthenticationConfig( "authCfg_Id_to_remove" )
```

13.3.4 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 Config*s, 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.

> **Nota:** 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:
>
> ```
> In [19]: authM = QgsAuthManager.instance()
> In [20]: authM.authMethod("Identity-Cert").supportedDataProviders()
> Out[20]: ['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(quri.encodedUri(), '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.

> **Aviso:** 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:

```
rlayer = QgsRasterLayer( quri.uri(False), 'states', 'wms')
```

> **Nota:** 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.

13.4 Adapt plugins to use Authentication infrastructure

Many third party plugins are using httplib2 to create HTTP connections instead of integrating with `QgsNetworkAccessManager` and its related Authentication Infrastructure integration. To facilitate this integration an 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
```

13.5 Authentication GUIs

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

13.5.1 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 <qgis.gui.QgsAuthConfigSelect>*.
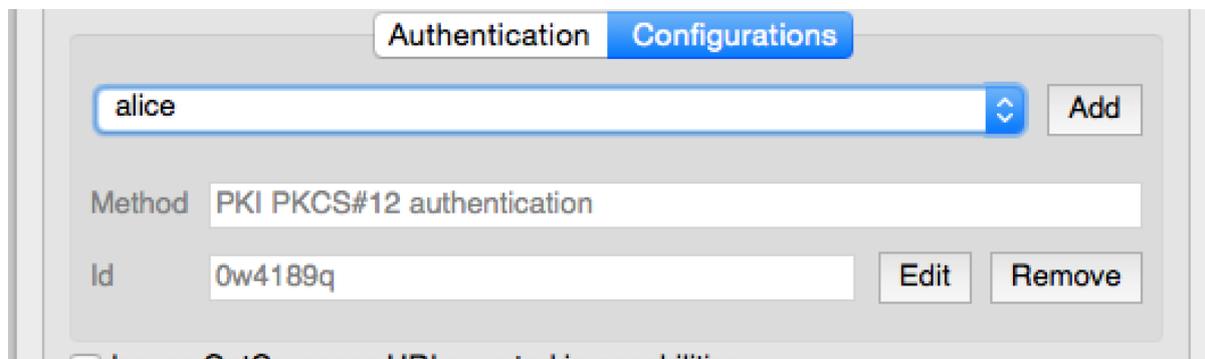
and can be used as in the following snippet:

```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with `parent`
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
# GUI has to be integrated
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.

13.5.2 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`
gui = QgsAuthConfigSelect( parent )
gui.show()
```

an integrated example can be found in the related test.

13.5.3 Authorities Editor GUI

A GUI used to manage only authorities is managed by the *QgsAuthAuthoritiesEditor <qgis.gui.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`
gui = QgsAuthAuthoritiesEditor( parent )
gui.show()
```
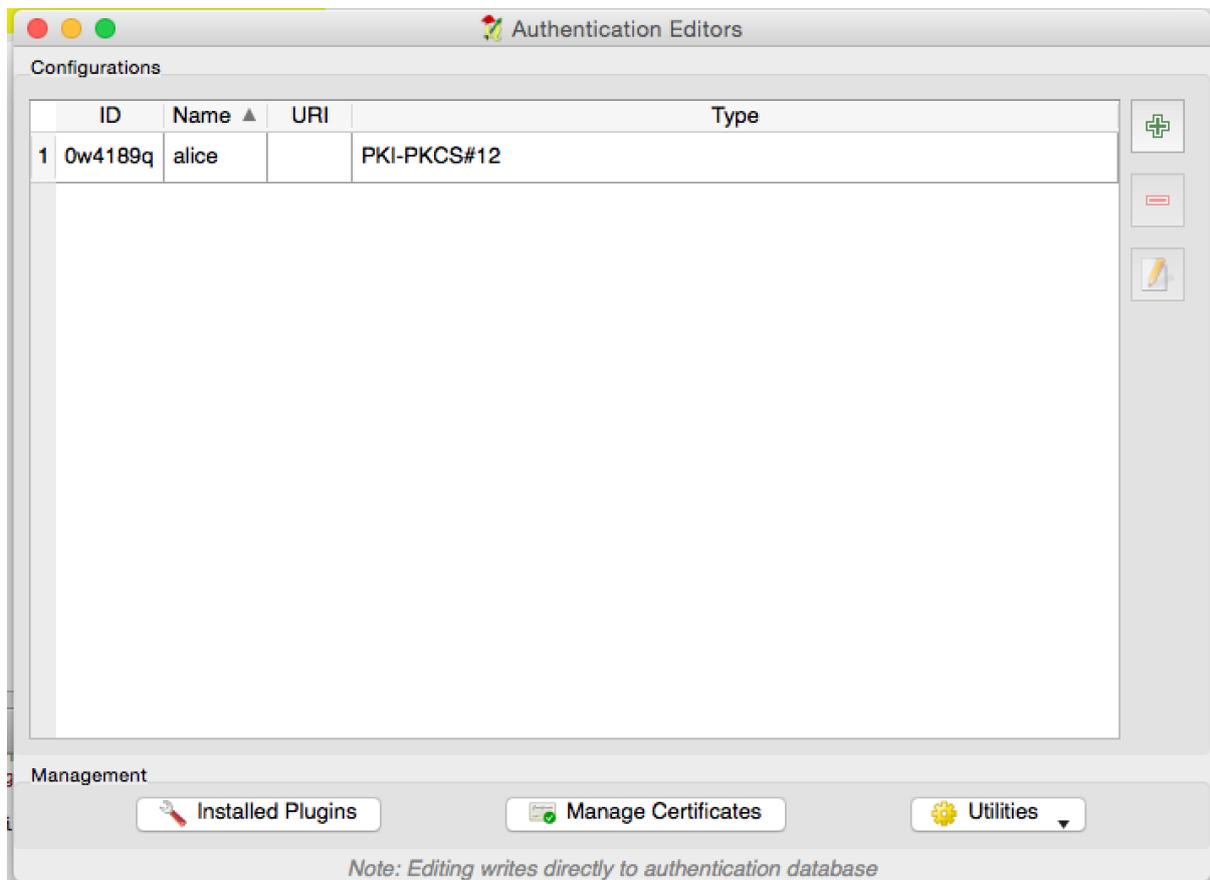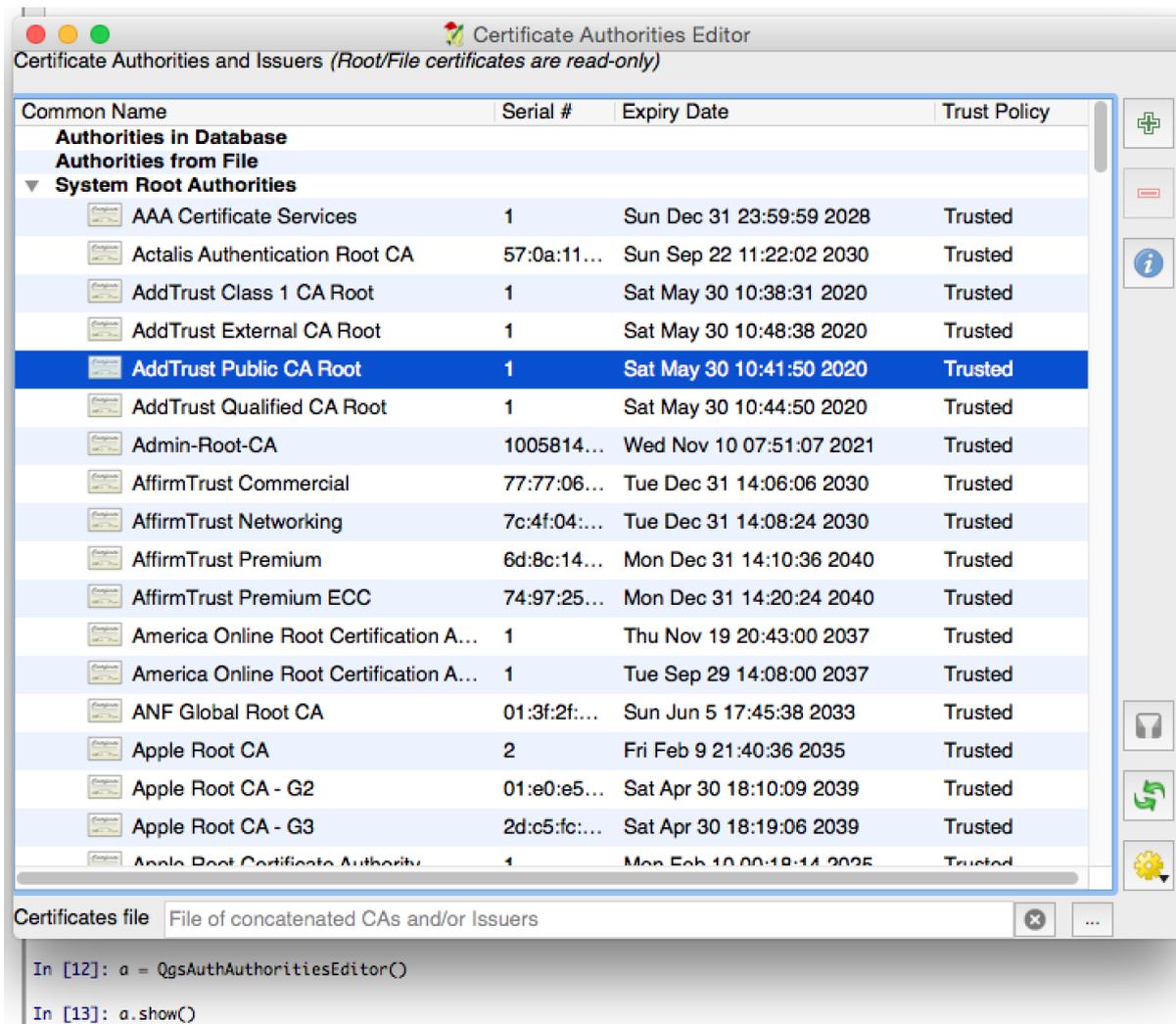
Tasks - doing heavy work in the background

14.1 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):
```

- Create a task from a function

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

- Create a task from a processing algorithm

```
QgsProcessingAlgRunnerTask('native:buffer', params, context,
                           feedback)
```

Aviso: Any background task (regardless of how it is created) must NEVER perform any GUI based operations, such as creating new widgets or interacting with existing widgets. Qt widgets must only be accessed or modified from the main thread. 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.

14.2 Examples

14.2.1 Extending QgsTask

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

```python
import random
from time import sleep

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

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""
    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None
    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
        Raising exceptions will crash QGIS, so we handle them
        internally and raise them in self.finished
        """
        QgsMessageLog.logMessage('Started task "{}"'.format(
                                     self.description()),
                                 MESSAGE_CATEGORY, Qgis.Info)
        wait_time = self.duration / 100
        for i in range(100):
            sleep(wait_time)
            # use setProgress to report progress
            self.setProgress(i)
            arandominteger = random.randint(0, 500)
            self.total += arandominteger
            self.iterations += 1
            # check isCanceled() to handle cancellation
            if self.isCanceled():
                return False
```

```python
                # 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(
                'Task "{name}" completed\n' \
                'Total: {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(
                    'Task "{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(
                    'Task "{name}" Exception: {exception}'.format(
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception
    def cancel(self):
        QgsMessageLog.logMessage(
            'Task "{name}" was canceled'.format(
                name=self.description()),
            MESSAGE_CATEGORY, Qgis.Info)
        super().cancel()


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

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

```
longtask.addSubTask(shortestsubtask)

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

14.2.2 Task from function

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

```python
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

def doSomething(task, wait_time):
    """
    Raises an exception to abort the task.
    Returns a result if success.
    The result will be passed, together with the exception (None in
    the case of success), to the on_finished method.
    If there is an exception, there will be no result.
    """
    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
                             MESSAGE_CATEGORY, Qgis.Info)
    wait_time = wait_time / 100
    total = 0
    iterations = 0
    for i in range(100):
        sleep(wait_time)
        # use task.setProgress to report progress
        task.setProgress(i)
        arandominteger = random.randint(0, 500)
        total += arandominteger
        iterations += 1
        # check task.isCanceled() to handle cancellation
        if task.isCanceled():
            stopped(task)
            return None
        # raise an exception to abort the task
        if arandominteger == 42:
            raise Exception('bad value!')
    return {'total': total, 'iterations': iterations,
            'task': task.description()}

def stopped(task):
    QgsMessageLog.logMessage(
        'Task "{name}" was canceled'.format(
            name=task.description()),
        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

# Creae 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)
```

14.2.3 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://www.opengis.ch/2018/06/22/threads-in-pyqgis3/.

Developing Python Plugins

15.1 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*: The essentials: a starting point `__init__.py`; fill in the *Plugin metadata* `metadata.txt`. Then implement your own design. A main Python plugin body e.g. `mainplugin.py`. Probably a form in Qt Designer `form.ui`, with its `resources.qrc`.

3. *Write code*: Write the code inside the `mainplugin.py`

4. *Test*: Close and re-open QGIS and import your plugin again. 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".

15.1.1 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 an online automated 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 and doesn't require an internet connection. This is the recommended option, as it produces 3.x compatible sources.

> **Aviso:** 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*

15.1.2 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.

Nota: 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*
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*

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 the 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
```

Nota: 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
```

__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

```python
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`)

```python
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):
    # 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:

```python
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)

```xml
<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
```

Nota: 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 specifying 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.

15.1.3 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.

15.1.4 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.

> **Aviso:** 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 just open QGIS, change the language (*Settings → Options → Language*) and restart QGIS.

You should see your plugin in the correct language.

> **Aviso:** 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.

15.1.5 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 as an experimental plugin 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.

15.2 Code Snippets

- *How to call a method by a key shortcut*
- *How to toggle Layers*
- *How to access attribute table of selected features*

This section features code snippets to facilitate plugin development.

15.2.1 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

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

15.2.2 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

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

15.2.3 How to access attribute table of selected features

```python
def change_value(value):
    """Change the value in the second column for all selected features.

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

The method requires one parameter (the new value for the second field of the selected feature(s)) and can be called by

```
changeValue(50)
```

15.3 Using Plugin Layers

> **Aviso:** *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.*

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.

TODO: Check correctness and elaborate on good use cases for QgsPluginLayer, . . .

15.3.1 Subclassing QgsPluginLayer

Below is an example of a minimal QgsPluginLayer implementation. It is an excerpt of the Watermark example plugin

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

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

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

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

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

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

You can also add code for displaying custom information in the layer properties

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

15.4 IDE settings for writing and debugging plugins

Aviso: *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.*

- *A note on configuring your IDE on Windows*
- *Debugging using Eclipse and PyDev*
  - *Installation*

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.

15.4.1 A note on configuring your IDE on Windows

On Linux there is no additional configuration needed to develop plugins. But on Windows you need to make sure you 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`.

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

• Make a copy of `qgis-unstable.bat` and rename it `pyscripter.bat`.

• Open it in an editor. And remove the last line, the one that starts QGIS.

• Add a line that points to your Pyscripter executable and add the command line argument that sets the version of Python to be used

• 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
```

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 sections, we will be explaining how to configure it for developing and testing plugins. To prepare your environment for using Eclipse in Windows, you should also create a batch file and use it to start Eclipse.

To create that batch file, follow these steps:

• 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`

• Locate your `eclipse.exe` executable.

• 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
```

15.4.2 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

Preparing QGIS

There is some preparation to be done on QGIS itself. Two plugins are of interest: **Remote Debug** and **Plugin reloader**.

- Go to *Plugins → Manage and Install plugins. . .*
- Search for *Remote Debug* (at the moment it's still experimental, so enable experimental plugins under the *Options* tab in case it does not show up). Install it.
- Search for *Plugin reloader* and install it as well. This will let you reload a plugin instead of having to close and restart QGIS to have the plugin reloaded.

Setting up Eclipse

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.

Now right-click your new project and choose *New → Folder*.

Click *Advanced* and choose *Link to alternate location (Linked Folder)*. In case you already have sources you want to debug, choose these. In case you don't, create a folder as it was already explained.

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

Configuring the debugger

To get the debugger working, switch to the Debug perspective in Eclipse (*Window → Open Perspective → Other → Debug*).

Now start the PyDev debug server by choosing *PyDev → Start Debug Server*.

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).

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.

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. 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. 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.
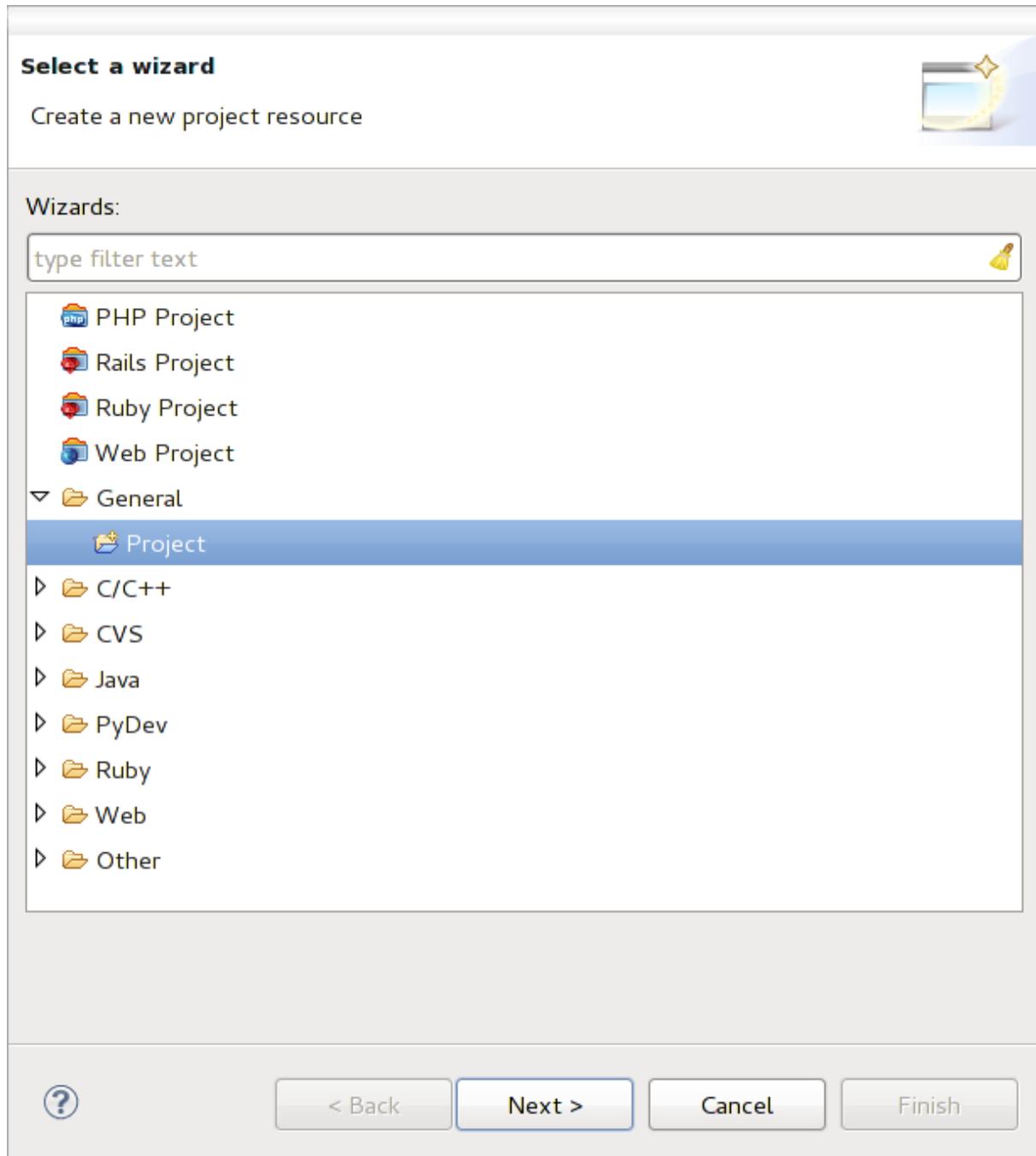
Figure 15.1: Eclipse project

```
         88
         89⊖        def printProfile(self):
    🔖   90            printer = QPrinter( QPrinter.HighResolution )
         91            printer.setOutputFormat( QPrinter.PdfFormat )
         92            printer.setPaperSize( QPrinter.A4 )
         93            printer.setOrientation( QPrinter.Landscape )
         94
         95            printPreviewDlg = QPrintPreviewDialog( )
         96            printPreviewDlg.paintRequested.connect( self.printRequested )
         97
         98            printPreviewDlg.exec_()
         99
        100        @pyqtSlot( QPrinter )
        101⊖       def printRequested( self, printer ):
        102            self.webView.print_( printer )
```
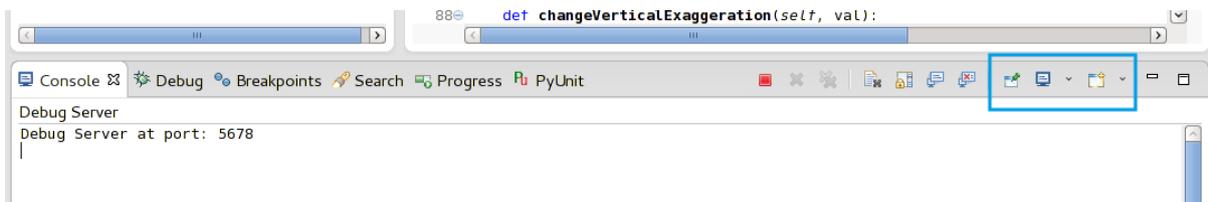
Figure 15.2: Breakpoint

Figure 15.3: PyDev Debug Console

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

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.

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*.

First open the Libraries tab. 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) open QGIS, start a python console and simply enter `qgis` and press Enter. It will show you which QGIS module it uses and its path. Strip the trailing `/qgis/__init__.pyc` from this path and you've got the path you are looking for.

You should also add your plugins folder here (it is in `python/plugins` under the user profile folder).

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 PyQt4 API. Therefore also add PyQt4 as forced builtin. That should probably already be present in your libraries tab.

Click *OK* and you're done.

Nota: 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.
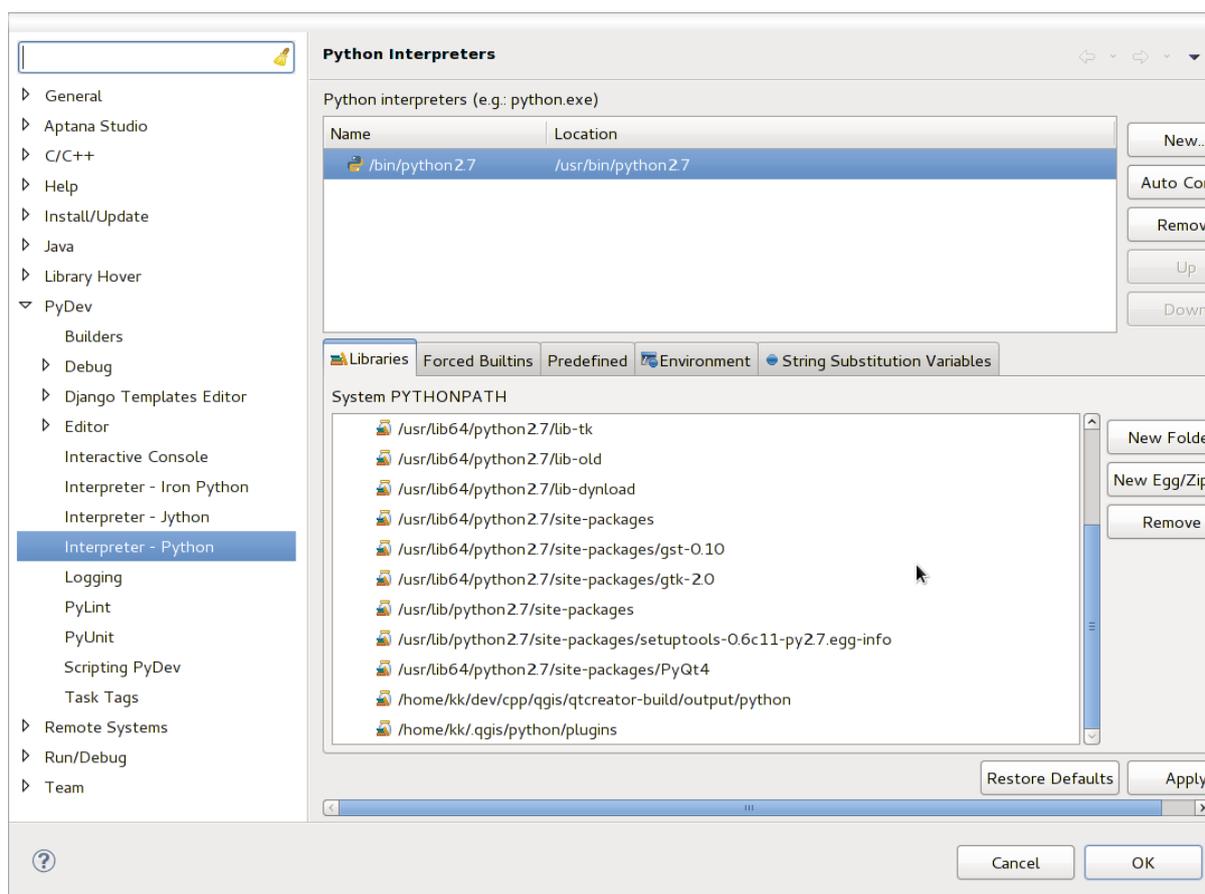

Figure 15.4: PyDev Debug Console

15.4.3 Debugging using PDB

If you do not use an IDE such as Eclipse, you can debug using PDB, following these steps.

First add this code in the spot where you would like to debug

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

Then run QGIS from the command line.

On Linux do:

```
$ ./Qgis
```

On macOS do:

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

And when the application hits your breakpoint you can type in the console!

TODO: Add testing information

15.5 Releasing your plugin

> **Aviso:** *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.*

- *Metadata and names*
- *Code and help*
- *Official Python plugin repository*
    - *Permissions*
    - *Trust management*
    - *Validation*
    - *Plugin structure*

Once your plugin is ready and you think the plugin could be helpful for some people, do not hesitate to upload it to *Official Python plugin repository*. On that page you can also find packaging guidelines about how to prepare the plugin to work well with the plugin installer. Or in case you would like to set up your own plugin repository, create a simple XML file that will list the plugins and their metadata.

Please take special care to the following suggestions:

15.5.1 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

15.5.2 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.

15.5.3 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.

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

Escrevendo um complemento de processamento

Depending on the kind of plugin that you are going to develop, it might be 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.

To create a plugin from scratch which contains an algorithm provider, you can follow these steps using the Plugin Builder:

- Instale o complemento Plugin Builder

- Crie um novo complemento usando o Plugin Builder. Quando o Plugin Builder perguntar qual modelo usar, selecione "Processing provider".

- 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.

If you want to add your existing plugin to Processing, you need to add some code.

In your `metadata.txt`, you need to add a variable:

```
hasProcessingProvider=yes
```

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 import Provider


class YourPluginName():

    def __init__(self):
        self.provider = None

    def initProcessing(self):
        self.provider = Provider()
        QgsApplication.processingRegistry().addProvider(self.provider)
```

```
    def initGui(self):
        self.initProcessing()

    def unload(self):
        QgsApplication.processingRegistry().removeProvider(self.provider)
```

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.

```python
from qgis.core import QgsProcessingProvider

from .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: https://github.com/qgis/QGIS/blob/release-3_4/python/plugins/processing/script/ScriptTemplate.py

Now you can reload your plugin in QGIS and you should see your example script in the Processing toolbox and modeler.

Biblioteca de análise de rede

> **Aviso:** *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.*

- *Informação Geral*
- *Elaborando um gráfico*
- *Análise de Gráficos*
    - *Encontrando os caminhos mais curtos*
    - *Areas of availability*

Starting from revision ee19294562 (QGIS >= 1.8) the new network analysis library was added to the QGIS core analysis library. The library:

- criar gráfico matemático para dado geográfico (camadas vetor polígono)
- implements 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.

17.1 Informação Geral

Resumidamente, um caso típico pode ser descrito assim:

1. criar gráfico para geodata (camada vetorial polígono usual)
2. rodar análise gráfica
3. usar resultados das análises (por exemplo, visualizá-los)

17.2 Elaborando um gráfico

A primeira coisa que você deve fazer — é preparar os dados de entrada, que é converter uma camada vetor em um gráfico. Todas as próximas ações irão usar este gráfico e não a camada.

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: QgsLineVectorLayerDirector. 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 QgsDistanceArcProperter strategy is available, that takes into account the length of the route. 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 networkanalysis module

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

Then some examples for creating a director

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

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

To construct a director we should pass a vector layer, that will be used as the source for the graph structure and information about allowed movement on each road segment (one-way or bidirectional movement, direct or reverse direction). The call looks like this

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

Aqui segue uma lista de todos os significados dos parâmetros:

- `vl` — 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. An integer. `1` indicates direct direction, `2` indicates reverse direction, and `3` indicates both directions.

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

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy

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

Now we can use the builder, which will create the graph. The `QgsGraphBuilder` class constructor takes several arguments:

- src — sistema de referência de coordenadas para uso. Argumento prioritário.

- otfAtivado — uso ou não da reprojeção "voo livre". Por padrão consta: *Verdadeiro* (use OTF).

- tolerânciaTopológica — tolerância topológica. Valor padrão é 0.

- elipsóideID — elipsóide usado. Por padrão "WGS84".

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

Também podemos definir vários pontos, que serão usados na análise. Por exemplo

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

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])
```

17.3 Análise de Gráficos

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 — tree) with the following properties:

- only one vertex has no incoming edges — the root of the tree

- todos os outros vértices têm apenas uma borda chegando

- 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:

- fonte — gráfico de entrada

- 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 i contains index of the incoming edge or -1 if there are no incoming edges. In the second array element i contains distance from the root of the tree to vertex i or DOUBLE_MAX if vertex i 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).

> **Aviso:** Use this code only as an example, it creates a lot of `QgsRubberBand` objects and may be slow on large datasets.

```python
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

Same thing but using the `dijkstra` method

```python
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

17.3.1 Encontrando os caminhos mais curtos

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

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 will need to select linestring layer in TOC and replace coordinates in the code with yours) that uses the `shortestTree` method

```python
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
```

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

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

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

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

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

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

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

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

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

And here is the same sample but using the `dijkstra` method

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

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

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

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

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

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

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

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

  p.append(tStart)

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

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

17.3.2 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.

Aqui está um exemplo

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

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

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

rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
rb.setColor(Qt.green)
```

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

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

idStart = graph.findVertex(tStart)

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

upperBound = []
r = 2000.0
i = 0
while i < len(cost):
  if cost[i] > r and tree[i] != -1:
    outVertexId = graph.arc(tree [i]).outVertex()
    if cost[outVertexId] < r:
      upperBound.append(i)
  i = i + 1

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

Complementos de servidores Python do QGIS

Aviso: *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.*

> – *allowToEdit*
>
> – *cacheKey*

Python plugins can also run on QGIS Server (see label_qgisserver):

- By using the *server interface* (`QgsServerInterface`) a Python plugin running on the server can alter the behavior of existing core services (**WMS**, **WFS** etc.).

- With the *server filter interface* (`QgsServerFilter`) you can change the input parameters, change the generated output or even provide new services.

- With the *access control interface* (`QgsAccessControlFilter`) you can apply some access restriction per requests.

18.1 Server Filter Plugins architecture

Server python plugins are loaded once when the FCGI application starts. They register one or more `QgsServerFilter` (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 a pseudo code showing a typical server session and when the filter's callbacks are called:

- **Get the incoming request**

    – create GET/POST/SOAP request handler

    – pass request to an instance of `QgsServerInterface`

    – call plugins `requestReady` filters

    – **if there is not a response**

        * **if SERVICE is WMS/WFS/WCS**

            · **criar servidor WMS/WFS/WCS**

                call server's `executeRequest` and possibly call `sendResponse` plugin filters when streaming output or store the byte stream output and content type in the request handler

        * call plugins `responseComplete` filters

    – call plugins `sendResponse` filters

    – request handler output the response

The following paragraphs describe the available callbacks in details.

18.1.1 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).

18.1.2 sendResponse

This is called whenever 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.

18.1.3 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).

18.2 Raising exception 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.

18.3 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 also access to a `QgsServerInterface`.

To tell QGIS Server that a plugin has a server interface, a special metadata entry is needed (in *metadata.txt*)

```
server=True
```

The example plugin discussed here (with many more example filters) is available on github: QGIS HelloServer Example Plugin. You could also find more examples at https://github.com/elpaso/qgis3-server-vagrant/tree/master/resources/web/plugins or browsing the QGIS plugins repository.

18.3.1 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*
```

18.3.2 __init__.py

This file is required by Python's import system. Also, QGIS Server requires that this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin `__init__.py` looks like:

```python
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from HelloServer import HelloServerServer
    return HelloServerServer(serverIface)
```

18.3.3 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 objects called QgsServerFilter.

Each `QgsServerFilter` implements one or more of the following callbacks:

- `requestReady()`
- `responseComplete()`
- `sendResponse()`

The following example implements a minimal filter which prints *HelloServer!* in case the **SERVICE** parameter equals to "HELLO":

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

class HelloFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(HelloFilter, self).__init__(serverIface)

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
            request.clearHeaders()
            request.setHeader('Content-type', 'text/plain')
            request.clearBody()
            request.appendBody('HelloServer!')
```

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

```
class HelloServerServer:
    def __init__(self, serverIface):
        # Save reference to the QGIS server interface
        self.serverIface = 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:

18.3.4 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:

```
from qgis.server import *
from qgis.core import *

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",
→'plugin', QgsMessageLog.INFO)
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete",
→'plugin', QgsMessageLog.CRITICAL)
```

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/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is:
→SERVICE=HELLO&request=GetOutput
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms]
→inserting pair SERVICE // HELLO into the parameter map
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms]
→inserting pair REQUEST // GetOutput into the parameter map
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter
→plugin default requestReady called
```

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪HelloFilter.requestReady
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default␣
↪configuration file path: /home/xxx/apps/bin/admin.sld
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking␣
↪byte array is ok to set...
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array␣
↪looks good, setting response...
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪HelloFilter.responseComplete
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪SUCCESS - ParamsFilter.responseComplete
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪RemoteConsoleFilter.responseComplete
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP␣
↪response
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪HelloFilter.sendResponse
```

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).

18.3.5 Modifying or replacing the output

The watermark filter example shows how to replace the WMS output with a new image obtained by adding a watermark image on the top of the WMS image generated by the WMS core service:

```python
import os

from qgis.server import *
from qgis.core import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *


class WatermarkFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(WatermarkFilter, self).__init__(serverIface)

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (request.parameter('SERVICE').upper() == 'WMS' \
                and request.parameter('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image␣
↪ready {}".format(request.infoFormat()), 'plugin', QgsMessageLog.INFO)
            # 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")
                # 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 returning PNG in any case.

18.4 Access control plugin

18.4.1 Plugin files

Here's the directory structure of our example server plugin:

```
PYTHON_PLUGINS_PATH/
  MyAccessControl/
    __init__.py    --> *required*
    AccessControl.py  --> *required*
    metadata.txt   --> *required*
```

18.4.2 __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
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from MyAccessControl.AccessControl import AccessControl
    return AccessControl(serverIface)
```

18.4.3 AccessControl.py

```python
class AccessControl(QgsAccessControlFilter):

    def __init__(self, server_iface):
        super(QgsAccessControlFilter, self).__init__(server_iface)

    def layerFilterExpression(self, layer):
        """ Return an additional expression filter """
        return super(QgsAccessControlFilter, self).layerFilterExpression(layer)

    def layerFilterSubsetString(self, layer):
        """ Return an additional subset string (typically SQL) filter """
        return super(QgsAccessControlFilter, self).layerFilterSubsetString(layer)
```

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

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

    def allowToEdit(self, layer, feature):
        """ Are we authorise to modify the following geometry """
        return super(QgsAccessControlFilter, self).allowToEdit(layer, feature)

    def cacheKey(self):
        return super(QgsAccessControlFilter, self).cacheKey()
```

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.

18.4.4 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".

18.4.5 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".

18.4.6 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.

Exemplo:

```
def layerPermissions(self, layer):
    rights = QgsAccessControlFilter.LayerPermissions()
    rights.canRead = True
    rights.canRead = rights.canInsert = rights.canUpdate = rights.canDelete = False
    return rights
```

To limit everything on read only access.

18.4.7 authorizedLayerAttributes

Used to limit the visibility of a specific subset of attribute.

The argument attribute return the current set of visible attributes.

Exemplo:

```
def authorizedLayerAttributes(self, layer, attributes):
    return [a for a in attributes if a != "role"]
```

To hide the 'role' attribute.

18.4.8 allowToEdit

This is used to limit the editing on a subset of features.

It is used in the `WFS-Transaction` protocol.

Exemplo:

```
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.

18.4.9 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.

Cheat sheet for PyQGIS

19.1 User Interface

Change Look & Feel

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
qss_file = open(r"/path/to/style/file.qss").read()
app.setStyleSheet(qss_file)
```

Change icon and title

```
from qgis.PyQt.QtGui import QIcon

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

19.2 Settings

Get QSettings list

```
from qgis.PyQt.QtCore import QSettings

qs = QSettings()

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

19.3 Toolbars

Remove toolbar

```
from qgis.utils import iface

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])
```

19.4 Menus

Remove menu

```
from qgis.utils import iface

# for example Help Menu
menu = iface.helpMenu()
menubar = menu.parentWidget()
menubar.removeAction(menu.menuAction())

# and add again
menubar.addAction(menu.menuAction())
```

19.5 Canvas

Access canvas

```
from qgis.utils import iface

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.PyQt.QtCore import QSettings
# Set milliseconds (150 milliseconds)
QSettings().setValue("/qgis/map_update_interval", 150)
```

19.6 Layers

Add vector layer

```python
from qgis.utils import iface

layer = iface.addVectorLayer("/path/to/shapefile/file.shp", "layer name you like",
→"ogr")
if not layer:
    print("Layer failed to load!")
```

Get active layer

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

List all layers

```python
from qgis.core import QgsProject

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

Obtain layers name

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

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

Otherwise

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

Find layer by name

```python
from qgis.core import QgsProject

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

Set active layer

```python
from qgis.core import QgsProject

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

Refresh layer at interval

```python
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

```python
dir(layer)
```

Adding new feature with feature form

```python
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

```python
from qgis.core import QgsPointXY

pr = layer.dataProvider()
feat = QgsFeature()
feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
pr.addFeatures([feat])
```

Get features

```python
for f in layer.getFeatures():
    print (f)
```

Get selected features

```python
for f in layer.selectedFeatures():
    print (f)
```

Get selected features Ids

```python
selected_ids = layer.selectedFeatureIds()
print(selected_ids)
```

Create a memory layer from selected features Ids

```python
from qgis.core import QgsFeatureRequest

memory_layer = layer.materialize(QgsFeatureRequest().setFilterFids(layer.
→selectedFeatureIds()))
QgsProject.instance().addMapLayer(memory_layer)
```

Get geometry

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

Move geometry

```python
geom.translate(100, 100)
poly.setGeometry(geom)
```

Set the CRS

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

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.
→EpsgCrsId))
```

See the CRS

```python
from qgis.core import QgsProject

for layer in QgsProject.instance().mapLayers().values():
    crs = layer.crs().authid()
    layer.setName('{} ({})'.format(layer.name(), crs))
```

Hide a field column

```python
from qgis.core import QgsEditorWidgetSetup

def fieldVisibility (layer,fname):
    setup = QgsEditorWidgetSetup('Hidden', {})
    for i, column in enumerate(layer.fields()):
        if column.name()==fname:
            layer.setEditorWidgetSetup(idx, setup)
            break
        else:
            continue
```

Layer from WKT

```python
from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject

layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.0934.89,-88.39 30.34,-89.57
↪30.18,-89.73 31,-91.63 30.99,-90.8732.37,-91.23 33.44,-90.93 34.23,-90.30 34.99,-
↪88.82 34.99))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])
```

Load all layers from GeoPackage

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

fileName = "/path/to/gpkg/file.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)

```python
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()
```

19.7 Table of contents

Access checked layers

```python
from qgis.utils import iface

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

Remove contextual menu

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

19.8 Advanced TOC

Root node

```python
from qgis.core import QgsProject

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

Access the first child node

```python
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))
```

Find groups and nodes

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

for child in root.children():
    if isinstance(child, QgsLayerTreeGroup):
        print ("- group: " + child.name())
    elif isinstance(child, QgsLayerTreeLayer):
        print ("- layer: " + child.name())
```

Find group by name

```python
print (root.findGroup("My Group"))
```

Add layer

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

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
```

Add group

```python
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
```

Remove layer

```python
root.removeLayer(layer1)
```

Remove group

```python
root.removeChildNode(node_group2)
```

Move node

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

Rename node

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

Changing visibility

```python
print (cloned_group1.isVisible())
cloned_group1.setItemVisibilityChecked(False)
node_layer1.setItemVisibilityChecked(False)
```

Expand node

```python
print (cloned_group1.isExpanded())
cloned_group1.setExpanded(False)
```

Hidden node trick

```python
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):
    print ("ADDED", node, indexFrom, indexTo)

root.willAddChildren.connect(onWillAddChildren)
root.addedChildren.connect(onAddedChildren)
```

Create new table of contents (TOC)

```
from qgis.core import QgsProject, QgsLayerTreeModel
from qgis.gui import QgsLayerTreeView

root = QgsProject.instance().layerTreeRoot()
model = QgsLayerTreeModel(root)
view = QgsLayerTreeView()
view.setModel(model)
view.show()
```

19.9 Processing algorithms

Get algorithms list

```
from qgis.core import QgsApplication

for alg in QgsApplication.processingRegistry().algorithms():
    print("{}:{} --> {}".format(alg.provider().name(), alg.name(), alg.
 displayName())))
```

Otherwise

```
def alglist():
    s = ''
    for i in QgsApplication.processingRegistry().algorithms():
        l = i.displayName().ljust(50, "-")
        r = i.id()
        s += '{}--->{}\n'.format(l, r)
    print(s)
```

Get algorithms help

Random selection

```
import processing

processing.algorithmHelp("qgis:randomselection")
```

Run the algorithm

For this example, the result is stored in a temporary memory layer which is added to the project.

```
import processing
result = processing.run("native:buffer", {'INPUT': layer, 'OUTPUT': 'memory:'})
QgsProject.instance().addMapLayer(result['OUTPUT'])
```

How many algorithms are there?

```
from qgis.core import QgsApplication

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

How many providers are there?

```python
from qgis.core import QgsApplication

len(QgsApplication.processingRegistry().providers())
```

How many expressions are there?

```python
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

19.10 Decorators

CopyRight

```python
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, x_offset, y_offset):
    p.translate( xOffset , yOffset  )
    text.drawContents(p)
    p.setWorldTransform( p.worldTransform() )

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

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

    # Calculate positions
    if case == 0:
        # Top Left
```

```
            add_copyright(p, text, xOffset, yOffset)

    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()
```

19.11 Sources

- QGIS Python (PyQGIS) API

- QGIS C++ API

- StackOverFlow QGIS questions

- Script by Klas Karlsson

- Boundless lib-qgis-common repository