PyQGIS 3.10 developer cookbook

QGIS Project

09 dez. 2020

Sumário

1 Introdução **1**
- 1.1 Escrevendo Scripts no Terminal Python . . . 2
- 1.2 Complementos Python . . . 2
- 1.3 Running Python code when QGIS starts . . . 3
  - 1.3.1 The `startup.py` file . . . 3
  - 1.3.2 The PYQGIS_STARTUP environment variable . . . 3
- 1.4 Aplicações Python . . . 3
  - 1.4.1 Using PyQGIS in standalone scripts . . . 4
  - 1.4.2 Using PyQGIS in custom applications . . . 4
  - 1.4.3 Executando aplicativos personalizados . . . 5
- 1.5 Technical notes on PyQt and SIP . . . 6

2 Carregando projetos **7**
- 2.1 Resolving bad paths . . . 8

3 Carregando Camadas **9**
- 3.1 Camadas Vetoriais . . . 9
- 3.2 Camadas Matriciais . . . 12
- 3.3 Instância QgsProject . . . 14

4 Acessando o sumário (TOC) **17**
- 4.1 A classe QgsProject . . . 17
- 4.2 Classe QgsLayerTreeGroup . . . 18

5 Usando Camadas Raster **21**
- 5.1 Detalhes da Camada . . . 21
- 5.2 Renderizador . . . 22
  - 5.2.1 Rasters com Banda Única . . . 22
  - 5.2.2 Rasters Multibandas . . . 23
- 5.3 Valores de Consulta . . . 23

6 Usando Camadas Vetoriais **25**
- 6.1 Recuperando informações sobre atributos . . . 26
- 6.2 Iterando sobre Camada Vetorial . . . 26
- 6.3 Selecionando características . . . 27
  - 6.3.1 Acessando atributos . . . 28
  - 6.3.2 Iteração sobre os feições selecionadas . . . 28
  - 6.3.3 Iterando sobre um subconjunto de feições . . . 29
- 6.4 Modificando Camadas Vetoriais . . . 30
  - 6.4.1 Adicionar feições . . . 30
  - 6.4.2 Excluir feições . . . 31

6.4.3 Modificar Feições . . . 31
6.4.4 Modificando Camadas Vetoriais com um Buffer . . . 31
6.4.5 Adicionando e Removendo Campos . . . 33
6.5 Utilizando Índices Espaciais . . . 33
6.6 Criando Camadas Vetoriais . . . 34
6.6.1 De uma instância de `QgsVectorFileWriter` . . . 34
6.6.2 Diretamente das feições . . . 36
6.6.3 De uma instância de `QgsVectorLayer` . . . 37
6.7 Aparencia (Simbologia) de Camadas de Vetor . . . 38
6.7.1 Renderizador de símbolo único . . . 39
6.7.2 Renderizador de Símbolo Categorizado . . . 40
6.7.3 Renderizador de Símbolo Graduado . . . 40
6.7.4 Trabalhando com Símbolos . . . 41
6.7.5 Criando Renderizadores Personalizados . . . 45
6.8 Outros Tópicos . . . 47

7 Manipulação Geométrica **49**
7.1 Construção de Geométria . . . 49
7.2 Acesso a Geometria . . . 50
7.3 Operações e Predicados Geométricos . . . 51

8 Suporte a projeções **55**
8.1 Sistemas de Referencia de Coordenadas . . . 55
8.2 Transformação de SRC . . . 56

9 Usando a Tela do Mapa **59**
9.1 Incorporar o Mapa da Tela . . . 60
9.2 Bandas raster e fazedor de vértices . . . 60
9.3 Usando Ferramentas de Mapas na Tela . . . 61
9.4 Desenhar ferramenta de mapa personalizada . . . 62
9.5 Desenhar itens da tela do mapa . . . 64

10 Renderização em impressão de mapas **67**
10.1 Renderização simples . . . 67
10.2 Renderizando camadas com CRS diferente . . . 68
10.3 Saída usando layout de impressão . . . 68
10.3.1 Exportando o layout . . . 70
10.3.2 Exportando um atlas como layout . . . 70

11 Expressões, filtragem e cálculo dos valores **71**
11.1 Expressões de Análise . . . 72
11.2 Expressões de Avaliação . . . 72
11.2.1 Expressões Básicas . . . 72
11.2.2 Expressões com características . . . 73
11.2.3 Filtrando uma camada com expressões . . . 74
11.3 Manipulando erros de expressão . . . 74

12 Leitura e Armazenamento de Configurações **77**

13 Comunicação com o usuário **81**
13.1 Mostrando mensagens. A classe QgsMessageBar . . . 81
13.2 Mostrando progresso . . . 84
13.3 Carregando . . . 84
13.3.1 QgsMessageLog . . . 85
13.3.2 The python built in logging module . . . 85

14 Infraestrutura de autenticação **87**
14.1 Introdução . . . 88
14.2 Glossário . . . 88

14.3 QgsAuthManager o ponto de entrada . . . 88
14.3.1 Inicie o gerente e defina a master password . . . 89
14.3.2 Preencha o authdb com uma nova entrada de Configuração de Autenticação . . . 89
14.3.3 Remover uma entrada de authdb . . . 91
14.3.4 Leave authcfg expansion to QgsAuthManager . . . 91
14.4 Adapte complementos para usar a infraestrutura de autenticação . . . 92
14.5 GUIs de autenticação . . . 92
14.5.1 GUI para selecionar credenciais . . . 92
14.5.2 GUI do Editor de Autenticação . . . 93
14.5.3 GUI do Editor de Autoridades . . . 94

15 Tarefas - trabalho pesado em segundo plano **95**
15.1 Introdução . . . 95
15.2 Exemplos . . . 96
15.2.1 Estendendo QgsTask . . . 96
15.2.2 Tarefa da função . . . 98
15.2.3 Tarefa de um algoritmo de processamento . . . 100

16 Desenvolvendo complementos Python **101**
16.1 Estruturando Complementos Python . . . 101
16.1.1 Escrevendo um complemento . . . 102
16.1.2 Conteúdo do complemento . . . 103
16.1.3 Documentação . . . 107
16.1.4 Tradução . . . 108
16.1.5 Dicas e truques . . . 110
16.2 Partes de código . . . 110
16.2.1 Como chamar um método por um atalho de teclas . . . 110
16.2.2 Como alternar Camadas . . . 111
16.2.3 Como acessar a tabela de atributos das feições selecionadas . . . 111
16.2.4 Interface for plugin in the options dialog . . . 111
16.3 Usando Camadas de Complementos . . . 113
16.3.1 Subclassificação QgsPluginLayer . . . 113
16.4 Configurações de IDE para gravar e depurar complementos . . . 114
16.4.1 Complementos úteis para escrever complementos Python . . . 114
16.4.2 Uma observação sobre como configurar seu IDE no Linux e Windows . . . 115
16.4.3 Depurando usando o Pyscripter IDE (Windows) . . . 115
16.4.4 Depurando usando Eclipse e PyDev . . . 115
16.4.5 Depurando com PyCharm no Ubuntu com um QGIS compilado . . . 120
16.4.6 Debugging using PDB . . . 122
16.5 Lançando seu complemento . . . 122
16.5.1 Metadados e nomes . . . 123
16.5.2 Código e ajuda . . . 123
16.5.3 Repositório Oficial de complementos Python . . . 123

17 Escrevendo um complemento de processamento **127**
17.1 Criando do zero . . . 127
17.2 Atualizando um complemento . . . 128

18 Biblioteca de análise de rede **131**
18.1 Informação Geral . . . 131
18.2 Elaborando um gráfico . . . 132
18.3 Análise de Gráficos . . . 134
18.3.1 Encontrando os caminhos mais curtos . . . 136
18.3.2 Areas of availability . . . 138

19 Servidor QGIS e Python **141**
19.1 Introdução . . . 141
19.2 Noções básicas da API do servidor . . . 142
19.3 Independente ou incorporado . . . 142

19.4 Complementos do servidor . . . 143
19.4.1 Complementos de filtro de servidor . . . 143
19.4.2 Custom services . . . 151
19.4.3 Custom APIs . . . 152

20 Folha de dicas para PyQGIS **155**
20.1 Interface de usuário . . . 155
20.2 Configurações . . . 155
20.3 Barra de ferramentas . . . 156
20.4 Menus . . . 156
20.5 Tela . . . 156
20.6 Camadas . . . 157
20.7 Table of contents . . . 160
20.8 Advanced TOC . . . 160
20.9 Processing algorithms . . . 164
20.10 Decorators . . . 164
20.11 Composer . . . 166
20.12 Sources . . . 166

CAPÍTULO 1

Introdução

Este documento pretende ser um tutorial e um guia de referência. Embora não liste todos os casos de uso possíveis, deve fornecer uma boa visão geral da funcionalidade principal.

- *Escrevendo Scripts no Terminal Python*
- *Complementos Python*
- *Running Python code when QGIS starts*
  - *The `startup.py` file*
  - *The PYQGIS_STARTUP environment variable*
- *Aplicações Python*
  - *Using PyQGIS in standalone scripts*
  - *Using PyQGIS in custom applications*
  - *Executando aplicativos personalizados*
- *Technical notes on PyQt and SIP*

O suporte ao Python foi introduzido pela primeira vez no QGIS 0.9. Existem várias maneiras de usar o Python no QGIS Desktop (abordado nas seções seguintes):

- Emita comandos no console Python no QGIS
- Crie e use complementos
- Executar automaticamente o código Python quando o QGIS for iniciado
- Create processing algorithms
- Create functions for expressions in QGIS
- Crie aplicativos personalizados com base na API QGIS

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

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

Um bom recurso para aprender como executar tarefas comuns é fazer o download de complementos existentes no repositório de complementos e examinar seu código.

1.1 Escrevendo Scripts no Terminal Python

O QGIS fornece um console Python integrado para scripts. Ele pode ser aberto no menu *Complementos → Terminal Python*

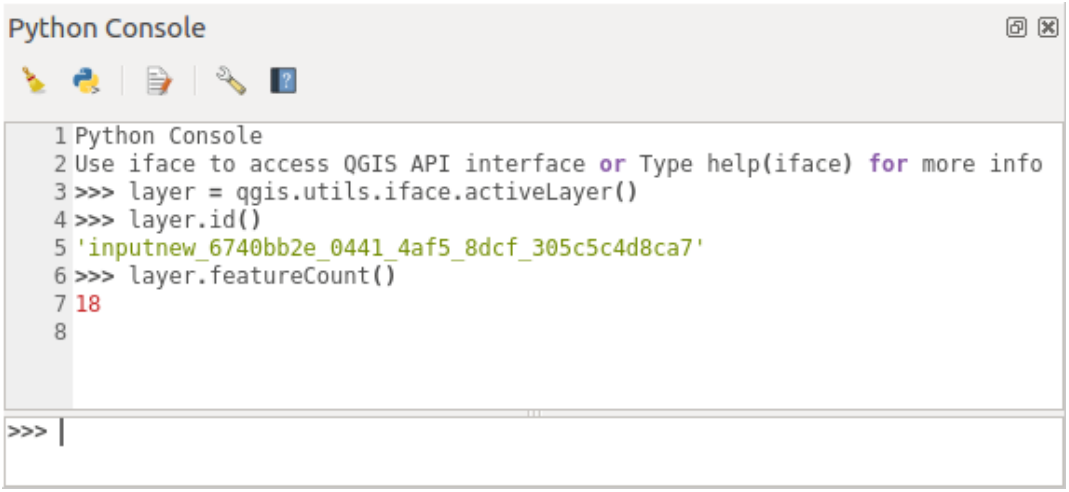


Fig. 1.1: Terminal Python QGIS

A captura de tela acima ilustra como obter a camada atualmente selecionada na lista de camadas, mostrar seu ID e, opcionalmente, se for uma camada vetorial, mostrar o número de feições. Para interação com o ambiente QGIS, existe uma variável `iface`, que é uma instância de `QgisInterface`. Essa interface permite o acesso à tela do mapa, menus, barras de ferramentas e outras partes do aplicativo QGIS.

Para conveniência do usuário, as seguintes instruções são executadas quando o terminal é iniciado (no futuro, será possível definir mais comandos iniciais)

```
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

A funcionalidade do QGIS pode ser estendida usando complementos. Os complementos podem ser escritos em Python. A principal vantagem sobre os complementos C++ é a simplicidade de distribuição (sem compilação para cada plataforma) e o desenvolvimento mais fácil.

Muitos complementos que abrangem várias funcionalidades foram escritos desde a introdução do suporte ao Python. O instalador do complemento permite aos usuários buscar, atualizar e remover facilmente os complementos do Python. Veja a página Python Plugins para obter mais informações sobre complementos e desenvolvimento de complementos.

Criar complementos no Python é simples, veja development_plugins para instruções detalhadas.

Nota: Os complementos Python também estão disponíveis para o servidor QGIS. Veja *Servidor QGIS e Python* para mais detalhes.

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 *Escrevendo Scripts no Terminal Python* 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
```

(continua na próxima página)

(continuação da página anterior)

```
19 # provider and layer registries from memory
20 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.

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

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

CAPÍTULO 2

Carregando projetos

Às vezes, você precisa carregar um projeto existente a partir de um complemento ou (com mais frequência) ao desenvolver um aplicativo QGIS Python independente (veja: *Aplicações Python*).

Para carregar um projeto no aplicativo QGIS atual, você precisa criar uma instância da classe `QgsProject`. Esta é uma classe singleton, portanto você deve usar o método `instance()` para fazer isso. Você pode chamar o método `read()`, passando o caminho do projeto a ser carregado:

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

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

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

Se você precisar fazer modificações no projeto (por exemplo, adicionar ou remover algumas camadas) e salvar suas alterações, chame o método `write()` da sua instância do projeto. O método `write()` também aceita um caminho opcional para salvar o projeto em um novo local:

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

Ambas as funções `read()` e `write()` retornam um valor booleano que você pode usar para verificar se a operação foi bem sucedida.

Nota: Se você estiver escrevendo um aplicativo independente do QGIS, para sincronizar o projeto carregado com a

tela, é necessário instanciar uma :class:QgsLayerTreeMapCanvasBridge <qgis.gui.QgsLayerTreeMapCanvasBridge> `como no exemplo abaixo:

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

```
...
```

2.1 Resolving bad paths

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

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

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

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

Some use cases:

1. replace an outdated path:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

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

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. replace stored database credentials with new ones:

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

CAPÍTULO 3

Carregando Camadas

Os trechos de código nesta página precisam das seguintes importações:

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

- *Camadas Vetoriais*
- *Camadas Matriciais*
- *Instância QgsProject*

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

Para criar e adicionar uma instância de camada vetorial ao projeto, especifique o identificador da fonte de dados da camada, o nome da camada e o nome do provedor:

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

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

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

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.

Para uma camada vetorial geopackage:

```
# get the path to a geopackage  e.g. /usr/share/qgis/resources/data/world_map.gpkg
path_to_gpkg = os.path.join(QgsApplication.pkgDataPath(), "resources", "data",
↪"world_map.gpkg")
# append the layername part
gpkg_countries_layer = path_to_gpkg + "|layername=countries"
# e.g. gpkg_places_layer = "/usr/share/qgis/resources/data/world_map.
↪gpkg|layername=countries"
vlayer = QgsVectorLayer(gpkg_countries_layer, "Countries layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")
else:
    QgsProject.instance().addMapLayer(vlayer)
```

A maneira mais rápida de abrir e exibir uma camada vetorial no QGIS é o método `addVectorLayer()` de `QgisInterface`:

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

Isso cria uma nova camada e a adiciona ao projeto QGIS atual (fazendo com que apareça na lista de camadas) em uma etapa. A função retorna a instância da camada ou `Nenhuma` se a camada não puder ser carregada.

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

- Biblioteca OGR (Shapefile e muitos outros formatos de arquivo) — fonte de dados é o caminho para o arquivo:
  - para Shapefile:

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

  - para dxf (observe as opções internas no uri da fonte de dados):

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

- Banco de dados PostGIS - fonte de dados é uma string com todas as informações necessárias para criar uma conexão com o banco de dados PostgreSQL.

  classe `QgsDataSourceUri` pode gerar essa string para você. Observe que o QGIS deve ser compilado com o suporte ao Postgres, caso contrário, esse provedor não estará disponível:

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

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

  Nota: O argumento `Falso` transmitido para `uri.uri(False)` impede a expansão dos parâmetros de configuração de autenticação, se você não estiver usando nenhuma configuração de autenticação, esse argumento não fará diferença.

- CSV ou outros arquivos de texto delimitados — para abrir um arquivo com ponto-e-vírgula como delimitador, com o campo "x" para a coordenada X e o campo "y" para a coordenada Y, você usaria algo como isto:

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

Nota: A string do provedor é estruturada como uma URL, portanto, o caminho deve ser prefixado com `file://`. Também permite geometrias no formato WKT (texto conhecido) como uma alternativa aos campos `x` e `y`, e permite que o sistema de referência de coordenadas seja especificado. Por exemplo:

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

- Arquivos GPX — o provedor de dados "gpx" lê trilhas, rotas e waypoints dos arquivos gpx. Para abrir um arquivo, o tipo (trilha/rota/waypoint) precisa ser especificado como parte do URL:

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

- Banco de dados SpatiaLite — Da mesma forma que os bancos de dados PostGIS, `QgsDataSourceUri` pode ser usado para geração do identificador da fonte de dados:

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

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

- Geometrias baseadas em MySQL WKB, através de OGR — fonte de dados são a string de conexão com a tabela:

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

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

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

O uri pode ser criado usando a biblioteca padrão `urllib`:

```
import urllib

params = {
    'service': 'WFS',
    'version': '1.1.0',
    'request': 'GetFeature',
    'typename': 'geosolutions:regioni',
```

(continua na próxima página)

(continuação da página anterior)

```
    'srsname': "EPSG:4326"
}
uri2 = 'https://demo.geo-solutions.it/geoserver/ows?' + urllib.parse.
↪unquote(urllib.parse.urlencode(params))
```

Nota: Você pode alterar a fonte de dados de uma camada existente chamando `setDataSource()` em uma instância `QgsVectorLayer`, como o exemplo a seguir:

```
uri = "https://demo.geo-solutions.it/geoserver/ows?service=WFS&version=1.1.0&
↪request=GetFeature&typename=geosolutions:regioni"
provider_options = QgsDataProvider.ProviderOptions()
# Use project's transform context
provider_options.transformContext = QgsProject.instance().transformContext()
vlayer.setDataSource(uri, "layer name you like", "WFS", provider_options)
QgsProject.instance().addMapLayer(vlayer)
```

3.2 Camadas Matriciais

Para acessar arquivos raster, a biblioteca GDAL é usada. Ele suporta uma ampla variedade de formatos de arquivo. Caso você tenha problemas para abrir alguns arquivos, verifique se o seu GDAL tem suporte para o formato específico (nem todos os formatos estão disponíveis por padrão). Para carregar um raster de um arquivo, especifique seu nome de arquivo e nome para exibição:

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

Para carregar um raster de um geopackage:

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

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

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

Da mesma forma que as camadas vetoriais, as camadas raster podem ser carregadas usando a função addRasterLayer do objeto `QgisInterface`:

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

Isso cria uma nova camada e a adiciona ao projeto atual (fazendo com que apareça na lista de camadas) em uma etapa.

To load a PostGIS raster:

PostGIS rasters, similar to PostGIS vectors, can be added to a project using a URI string. It is efficient to create a dictionary of strings for the database connection parameters. The dictionary is then loaded into an empty URI, before adding the raster. Note that None should be used when it is desired to leave the parameter blank:

```
uri_config = {#
# a dictionary of database parameters
'dbname':'gis_db', # The PostgreSQL database to connect to.
'host':'localhost',      # The host IP address or localhost.
'port':'5432',           # The port to connect on.
'sslmode':'disable',     # The SSL/TLS mode. Options: allow, disable, prefer,␣
↪require, verify-ca, verify-full
# user and password are not needed if stored in the authcfg or service
'user':None,             # The PostgreSQL user name, also accepts the new WFS␣
↪provider naming.
'password':None,         # The PostgreSQL password for the user.
'service':None,          # The PostgreSQL service to be used for connection to the␣
↪database.
'authcfg':'QconfigId',   # The QGIS athentication database ID holding connection␣
↪details.
# table and raster column details
'schema':'public',       # The database schema that the table is located in.
'table':'my_rasters',    # The database table to be loaded.
'column':'rast',         # raster column in PostGIS table
'mode':'2',              # GDAL 'mode' parameter, 2 union raster tiles, 1 separate␣
↪tiles (may require user input)
'sql':None,              # An SQL WHERE clause.
'key':None,              # A key column from the table.
'srid':None,             # A string designating the SRID of the coordinate␣
↪reference system.
'estimatedmetadata':'False', # A boolean value telling if the metadata is␣
↪estimated.
'type':None,             # A WKT string designating the WKB Type.
'selectatid':None,       # Set to True to disable selection by feature ID.
'options':None,          # other PostgreSQL connection options not in this list.
'connect_timeout':None,
'hostaddr':None,
'driver':None,
'tty':None,
'requiressl':None,
'krbsrvname':None,
'gsslib':None,
}
# configure the URI string with the dictionary
uri = QgsDataSourceUri()
for param in uri_config:
    if (uri_config[param] != None):
        uri.setParam(param, uri_config[param]) # add parameters to the URI

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

As camadas raster também podem ser criadas a partir de um serviço WCS:

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

Está aqui uma descrição dos parâmetros que o URI do WCS pode conter:

O WCS URI é composto por pares **chave=valor** separados por `&`. É o mesmo formato, como a string de consulta no URL, codificada da mesma maneira. `QgsDataSourceUri` deve ser usado para construir o URI para garantir que os caracteres especiais sejam codificados corretamente.

- **url** (obrigatório): URL do servidor WCS. Não use VERSION no URL, porque cada versão do WCS está usando um nome de parâmetro diferente para a versão **GetCapabilities**, consulte a versão param.

- **identifier** (obrigatório): Nome da cobertura
- **tempo** (opcional): posição ou período de tempo (beginPosition/endPosition[/timeResolution])
- **format** (opcional): nome do formato suportado. O padrão é o primeiro formato suportado com tif no nome ou o primeiro formato suportado.
- **crs** (opcional): SRC no formato AUTHORITY:ID, p. EPSG: 4326. O padrão é EPSG:4326 se suportado ou o primeiro SRC suportado.
- **username** (opcional): nome de usuário para autenticação básica.
- **password** (opcional) : Senha para autenticação básica.
- **IgnoreGetMapUrl** (opcional, hack): se especificado (definido como 1), ignore o URL GetCoverage anunciado por GetCapabilities. Pode ser necessário se um servidor não estiver configurado corretamente.
- **InvertAxisOrientation** (opcional, hack): se especificado (definido como 1), alterne o eixo na solicitação GetCoverage. Pode ser necessário para o SRC geográfico se um servidor estiver usando a ordem do eixo incorreta.
- **IgnoreAxisOrientation** (opcional, hack): Se especificado (definido como 1), não inverta a orientação do eixo de acordo com o padrão WCS para SRC geográfico.
- **cache** (opcional): controle de carregamento de cache, conforme descrito em QNetworkRequest::CacheLoadControl, mas a solicitação é reenviada como PreferCache se houver falha no AlwaysCache. Valores permitidos: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. O padrão é AlwaysCache.

Como alternativa, você pode carregar uma camada raster do servidor WMS. No entanto, atualmente não é possível acessar a resposta GetCapabilities da API - você precisa saber quais camadas deseja:

```
urlWithParams = "crs=EPSG:4326&format=image/png&layers=tasmania&styles&url=https://
↪demo.geo-solutions.it/geoserver/ows"
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print("Layer failed to load!")
```

3.3 Instância QgsProject

Se você deseja usar as camadas abertas para renderização, não se esqueça de adicioná-las à instância `QgsProject`. A instância `QgsProject` assume a propriedade das camadas e elas podem ser acessadas posteriormente de qualquer parte do aplicativo por seu ID exclusivo. Quando a camada é removida do projeto, ela também é excluída. As camadas podem ser removidas pelo usuário na interface QGIS ou via Python usando o método `removeMapLayer()`.

A adição de uma camada ao projeto atual é feita usando o método `addMapLayer()`:

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

Para adicionar uma camada em uma posição absoluta:

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

Se você deseja excluir a camada, use o método `removeMapLayer()`:

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

No código acima, o ID da camada é passado (você pode obtê-lo chamando o método `id()` da camada), mas você também pode passar o próprio objeto da camada.

Para obter uma lista de camadas carregadas e IDs de camada, use o método :meth: *mapLayers() <qgis.core.QgsProject.mapLayers>*:

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

CAPÍTULO 4

Acessando o sumário (TOC)

- *A classe QgsProject*
- *Classe QgsLayerTreeGroup*

Você pode usar diferentes classes para acessar todas as camadas carregadas no sumário e usá-las para recuperar informações:

- `QgsProject`
- `QgsLayerTreeGroup`

4.1 A classe QgsProject

Você pode usar `QgsProject` para recuperar informações sobre o sumário e todas as camadas carregadas.

Você precisa criar uma `instância` de `QgsProject` e usar seus métodos para obter as camadas carregadas.

O método principal é `mapLayers()`. Ele retornará um dicionário das camadas carregadas:

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

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

As `chaves` de dicionário são os IDs exclusivos da camada, enquanto os `valores` são os objetos relacionados.

Agora é fácil obter qualquer outra informação sobre as camadas:

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

(continua na próxima página)

(continuação da página anterior)

```
7
8 print(layers_list)
```

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

Você também pode consultar o sumário usando o nome da camada

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

Nota: Uma lista com todas as camadas correspondentes é retornada, portanto indexamos com `[0]` para obter a primeira camada com esse nome.

4.2 Classe QgsLayerTreeGroup

A árvore de camadas é uma estrutura de árvore clássica construída de nós. Atualmente, existem dois tipos de nós: nós de grupo (`QgsLayerTreeGroup`) e nós de camada (`QgsLayerTreeLayer`).

Nota: para mais informações, você pode ler essas postagens de Martin Dobias: Parte 1 Parte 2 Parte 3

A árvore da camada do projeto pode ser acessada facilmente com o método `layerTreeRoot()` da classe `QgsProject`:

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

`root` é um nó de grupo e tem *filhos*:

```
root.children()
```

Uma lista de filhos diretos é retornada. Os filhos do subgrupo devem ser acessados de seus próprios pais diretos.

Podemos recuperar um dos filhos:

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

```
<qgis._core.QgsLayerTreeLayer object at 0x7f1e1ea54168>
```

As camadas também podem ser recuperadas usando seu (único) `id`:

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

E os grupos também podem ser pesquisados usando seus nomes:

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

`QgsLayerTreeGroup` possui muitos outros métodos úteis que podem ser usados para obter mais informações sobre o sumário:

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

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

Agora vamos adicionar algumas camadas à árvore de camadas do projeto. Existem duas maneiras de fazer isso:

1. **Adição explícita** usando as funções `addLayer()` ou `insertLayer()`:

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

2. **Adição implícita**: como a árvore da camada do projeto está conectada ao registro da camada, basta adicionar uma camada ao registro da camada do mapa:

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

Você pode alternar entre `QgsVectorLayer` e `QgsLayerTreeLayer` facilmente:

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

```
Layer node: <qgis._core.QgsLayerTreeLayer object at 0x7fecceb46ca8>
Map layer: <QgsMapLayer: 'countries' (ogr)>
```

Grupos podem ser adicionados com o método `addGroup()`. No exemplo abaixo, o primeiro adicionará um grupo ao final do sumário, enquanto no último você poderá adicionar outro grupo dentro de um existente:

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

Para mover nós e grupos, existem muitos métodos úteis.

A movimentação de um nó existente é feita em três etapas:

1. clonando o nó existente
2. movendo o nó clonado para a posição desejada
3. excluindo o nó original

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

É um pouco mais *complicado* mover uma camada na legenda:

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

(continua na próxima página)

(continuação da página anterior)

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

ou movê-la para um grupo existente:

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

Alguns outros métodos que podem ser usados para modificar os grupos e camadas:

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

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

CAPÍTULO 5

Usando Camadas Raster

5.1 Detalhes da Camada

Uma camada raster consiste em uma ou mais bandas raster - conhecidas como rasters de banda única e multibanda. Uma banda representa uma matriz de valores. Uma imagem colorida (por exemplo, foto aérea) é um raster composto por faixas vermelhas, azuis e verdes. Rasters de banda única normalmente representam variáveis contínuas (por exemplo, elevação) ou variáveis discretas (por exemplo, uso do solo). Em alguns casos, uma camada raster vem com uma paleta e os valores raster referem-se às cores armazenadas na paleta.

O código a seguir assume que o objeto `rlayer` é uma `QgsRasterLayer`.

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

```
919 619
```

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

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

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

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

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

```
0
```

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

```
1
```

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

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

5.2 Renderizador

Quando uma camada raster é carregada, ela obtém um renderizador padrão com base em seu tipo. Pode ser alterado nas propriedades da camada ou usando programação.

Para consultar o renderizador atual:

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

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

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

```
singlebandgray
```

Para definir um renderizador, use o método `setRenderer` de `QgsRasterLayer`. Existem várias classes de renderizador (derivadas de `QgsRasterRenderer`):

- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`
- `QgsSingleBandColorDataRenderer`
- `QgsSingleBandGrayRenderer`
- `QgsSingleBandPseudoColorRenderer`

As camadas raster de banda única podem ser desenhadas nas cores cinza (valores baixos = preto, valores altos = branco) ou com um algoritmo de pseudo-cor que atribui cores aos valores. Rasters de banda única com uma paleta também podem ser desenhados usando a paleta. Camadas de multibandas geralmente são desenhadas mapeando as bandas para cores RGB. Outra possibilidade é usar apenas uma banda para desenhar.

5.2.1 Rasters com Banda Única

Digamos que queremos uma camada raster de renderização de banda única com cores que variam de verde a amarelo (correspondendo a valores de pixel de 0 a 255). No primeiro estágio, prepararemos um objeto `QgsRasterShader` e configuraremos sua função de sombreamento:

```
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)
```

Os sombreador mapeia as cores conforme especificado pelo seu mapa de cores. O mapa de cores é fornecido como uma lista de valores de pixels com cores associadas. Existem três modos de interpolação:

- linear (`Interpolated`): a cor é interpolada linearmente a partir das entradas do mapa de cores acima e abaixo do valor do pixel

- discreto (`Discrete`): a cor é obtida da entrada mais próxima do mapa de cores com valor igual ou superior
- exato (`Exact`): a cor não é interpolada, apenas pixels com valores iguais às entradas do mapa de cores serão desenhados

Na segunda etapa, associaremos esse sombreador à camada raster:

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

O número ``1 '' no código acima é o número da banda (as bandas raster são indexadas a partir de uma).

Finalmente, temos que usar o método `triggerRepaint` para ver os resultados:

```
rlayer.triggerRepaint()
```

5.2.2 Rasters Multibandas

Por padrão, o QGIS mapeia as três primeiras bandas para vermelho, verde e azul para criar uma imagem colorida (este é o estilo de desenho `MultiBandColor`. Em alguns casos, você pode substituir essas configurações. O código a seguir troca a faixa vermelha (1) e faixa verde (2):

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

No caso de apenas uma banda ser necessária para a visualização do raster, o desenho de banda única pode ser escolhido, tanto em níveis de cinza quanto em pseudocolor.

Temos que usar `triggerRepaint` para atualizar o mapa e ver o resultado:

```
rlayer_multi.triggerRepaint()
```

5.3 Valores de Consulta

Os valores raster podem ser consultados usando o método `sample` de `QgsRasterDataProvider`. Você deve especificar uma `QgsPointXY` e o número da banda da camada raster que você deseja consultar. O método retorna uma tupla com o valor e `True` ou `False`, dependendo dos resultados:

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

Outro método para consultar valores raster é usar o método `identify` que retorna um objeto `QgsRasterIdentifyResult`.

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

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

```
{1: 323.0}
```

Nesse caso, o método `results` retorna um dicionário, com índices de banda como chaves, e valores de banda como valores. Por exemplo, algo como `{1: 323.0}`

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

from qgis.core.additions.edit import edit

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

CAPÍTULO 6

Usando Camadas Vetoriais

- *Recuperando informações sobre atributos*
- *Iterando sobre Camada Vetorial*
- *Selecionando características*
  - *Acessando atributos*
  - *Iteração sobre os feições selecionadas*
  - *Iterando sobre um subconjunto de feições*
- *Modificando Camadas Vetoriais*
  - *Adicionar feições*
  - *Excluir feições*
  - *Modificar Feições*
  - *Modificando Camadas Vetoriais com um Buffer*
  - *Adicionando e Removendo Campos*
- *Utilizando Índices Espaciais*
- *Criando Camadas Vetoriais*
  - *De uma instância de* `QgsVectorFileWriter`
  - *Diretamente das feições*
  - *De uma instância de* `QgsVectorLayer`
- *Aparencia (Simbologia) de Camadas de Vetor*
  - *Renderizador de símbolo único*
  - *Renderizador de Símbolo Categorizado*
  - *Renderizador de Símbolo Graduado*
  - *Trabalhando com Símbolos*
    - *Trabalhando com Camadas de Símbolos*

        * *Criando Tipos de Camadas de Símbolos Personalizadas*
    - *Criando Renderizadores Personalizados*
- *Outros Tópicos*

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

A maioria dos trabalhos aqui é baseada nos métodos da classe `QgsVectorLayer`.

6.1 Recuperando informações sobre atributos

Você pode recuperar informações sobre os campos associados a uma camada vetorial chamando `fields()` em um objeto `QgsVectorLayer`:

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

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

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

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

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

```
NAME
```

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

6.2 Iterando sobre Camada Vetorial

A iteração sobre as feições em uma camada vetorial é uma das tarefas mais comuns. Abaixo está um exemplo do código básico simples para executar esta tarefa e mostrando algumas informações sobre cada feição. Supõe-se que a variável `layer` tenha um objeto `QgsVectorLayer`.

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

for feature in features:
    # retrieve every feature with its geometry and attributes
    print("Feature ID: ", feature.id())
    # fetch geometry
    # show some information about the feature geometry
    geom = feature.geometry()
```

(continua na próxima página)

(continuação da página anterior)

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

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

6.3 Selecionando características

Na área de trabalho do QGIS, as feições podem ser selecionadas de diferentes maneiras: o usuário pode clicar em uma feição, desenhar um retângulo na tela do mapa ou usar um filtro de expressão. As feições selecionadas normalmente são destacadas em uma cor diferente (o padrão é amarelo) para chamar a atenção do usuário na seleção.

Às vezes, pode ser útil selecionar recursos programaticamente ou alterar a cor padrão.

Para selecionar todas as feições, o método `selectAll()` pode ser usado:

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

Para selecionar usando uma expressão, use o método `selectByExpression()`:

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

Para alterar a cor da seleção, você pode usar o método `setSelectionColor()` de `QgsMapCanvas`, como mostrado no exemplo a seguir:

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

Para adicionar recursos à lista de recursos selecionados para uma determinada camada, você pode chamar `select()` passando para ele a lista de IDs de recursos:

```
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)
```

Para limpar a seleção:

```
layer.removeSelection()
```

6.3.1 Acessando atributos

Os atributos podem ser referidos pelo seu nome:

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

```
First feature
```

Como alternativa, os atributos podem ser referidos pelo índice. Isso é um pouco mais rápido do que usar o nome. Por exemplo, para obter o segundo atributo:

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

```
First feature
```

6.3.2 Iteração sobre os feições selecionadas

Se você precisar apenas de feições selecionadas, poderá usar o método `selectedFeatures()` da camada vetorial:

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

6.3.3 Iterando sobre um subconjunto de feições

Se você deseja iterar sobre um determinado subconjunto de feições em uma camada, como aquelas dentro de uma determinada área, você deve adicionar um objeto `QgsFeatureRequest` ao `getFeatures()` chamado. Aqui está um exemplo:

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

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

Por uma questão de velocidade, a interseção geralmente é feita apenas usando a caixa delimitadora da feição. No entanto, existe um sinalizador `ExactIntersect` que garante que apenas as feições que se intersectam serão retornadas:

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

Com `setLimit()` você pode limitar o número de feições solicitadas. Aqui está um exemplo:

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

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

Se você precisar de um filtro baseado em atributo em vez (ou adicionalmente) de um filtro espacial, como mostrado nos exemplos acima, poderá criar um objeto `QgsExpression` e passá-lo para o construtor **:classe:`QgsFeatureRequest <qgis.core.QgsFeatureRequest>`**. Aqui está um exemplo:

```
# 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)
```

Veja expression para obter detalhes sobre a sintaxe suportada por `QgsExpression`.

A solicitação pode ser usada para definir os dados recuperados para cada feição, portanto, o iterador retorna todos as feições, mas retorna dados parciais para cada uma delas.

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

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

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

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

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

6.4 Modificando Camadas Vetoriais

A maioria dos provedores de dados vetoriais suporta a edição de dados da camada. Às vezes, eles suportam apenas um subconjunto de possíveis ações de edição. Use a função `capacidades()` para descobrir qual conjunto de funcionalidades é suportado.

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

```
The layer supports DeleteFeatures
```

Para obter uma lista de todos os recursos disponíveis, consulte a `API Documentation of QgsVectorDataProvider`.

Para imprimir a descrição textual dos recursos da camada em uma lista separada por vírgulas, você pode usar `capabilitiesString()` como no exemplo a seguir:

```
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'
```

Ao usar qualquer um dos métodos a seguir para edição da camada vetorial, as alterações são confirmadas diretamente no armazenamento de dados subjacente (um arquivo, banco de dados etc.). Caso você deseje fazer apenas alterações temporárias, pule para a próxima seção que explica como fazer *modifications with editing buffer*.

Nota: Se você estiver trabalhando no QGIS (no console ou em um complemento), pode ser necessário forçar um redesenho da tela do mapa para ver as alterações feitas na geometria, no estilo ou nos atributos:

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

6.4.1 Adicionar feições

Crie algumas instâncias `QgsFeature` e passe uma lista delas para o método do provedor `addFeatures()` do provedor. Ele retornará dois valores: resultado (verdadeiro/falso) e lista de recursos adicionados (seu ID é definido pelo armazenamento de dados).

Para configurar os atributos da feição, você pode inicializar a feição passando um objeto `QgsFields` (você pode obtê-lo no método `fields()` da camada vetorial) ou chame `initAttributes()` passando o número de campos que você deseja adicionar.

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

6.4.2 Excluir feições

Para excluir algumas feições, basta fornecer uma lista de seus IDs de feições.

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

6.4.3 Modificar Feições

É possível alterar a geometria da feição ou alterar alguns atributos. O exemplo a seguir altera primeiro os valores dos atributos com os índices 0 e 1, depois altera a geometria da feição.

```
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: Preferindo a classe QgsVectorLayerEditUtils para edições somente de geometria

Se você precisar alterar apenas geometrias, considere usar `QgsVectorLayerEditUtils`, que fornece alguns métodos úteis para editar geometrias (traduzir, inserir ou mover vértices, etc.).

6.4.4 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 committed, all changes from the editing buffer are saved to data provider.

Os métodos são semelhantes aos que vimos no provedor, mas são chamados no objeto `QgsVectorLayer`.

Para que esses métodos funcionem, a camada deve estar no modo de edição. Para iniciar o modo de edição, use o método `startEditing()`. Para parar a edição, use os métodos `commitChanges()` ou `rollBack()`. O primeiro confirmará todas as suas alterações na fonte de dados, enquanto o segundo as descartará e não modificará a fonte de dados.

Para descobrir se uma camada está no modo de edição, use o método `isEditable()`.

Aqui você tem alguns exemplos que demonstram como usar esses métodos de edição.

```
from qgis.PyQt.QtCore import QVariant

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

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

(continua na próxima página)

(continuação da página anterior)

```
layer.deleteFeature(fid)

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

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

Para desfazer/refazer o trabalho corretamente, os chamados acima mencionados têm de ser envoltos em comandos de desfazer. (Se você não se importa com desfazer/refazer e quer ter as mudanças armazenadas imediatamente, então você vai ter o trabalho mais fácil *editing with data provider*.)

Aqui está como você pode usar a funcionalidade desfazer:

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

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

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

# ... more editing ...

layer.endEditCommand()
```

O método `beginEditCommand()` cria um comando interno "ativo" e registra as alterações subsequentes na camada vetorial. Com o chamado para `endEditCommand()` o comando é enviado para a pilha de desfazer e o usuário poderá desfazê-lo/refazê-lo da GUI. Caso algo dê errado ao fazer as alterações, o método `destroyEditCommand()` removerá o comando e reverterá todas as alterações feitas enquanto este comando estava ativo.

Você também pode usar o `with edit(layer)` -statement para agrupar commit e rollback em um bloco de código mais semântico, como mostrado no exemplo abaixo:

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

Isso chamará automaticamente `commitChanges()` no final. Se ocorrer alguma exceção, ele irá `rollBack()` todas as alterações. Caso seja encontrado um problema em `commitChanges()` (quando o método retornar Falso) a exceção a `QgsEditError` será gerada.

6.4.5 Adicionando e Removendo Campos

Para adicionar campos (atributos), você precisa especificar uma lista de definições de campos. Para exclusão de campos, forneça apenas uma lista de índices de campos.

```
from qgis.PyQt.QtCore import QVariant

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

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

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

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

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

Após adicionar ou remover campos no provedor de dados, os campos da camada precisam ser atualizados porque as alterações não são propagadas automaticamente.

```
layer.updateFields()
```

**Dica: Salvando as alterações diretamente usando o comando baseado em **``with`` **

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

6.5 Utilizando Índices Espaciais

Os índices espaciais podem melhorar drasticamente o desempenho do seu código, se você precisar fazer consultas frequentes a uma camada vetorial. Imagine, por exemplo, que você esteja escrevendo um algoritmo de interpolação e que, para um determinado local, precise conhecer os 10 pontos mais próximos de uma camada de pontos, a fim de usá-los para calcular o valor interpolado. Sem um índice espacial, a única maneira de o QGIS encontrar esses 10 pontos é calcular a distância de cada ponto até o local especificado e comparar essas distâncias. Isso pode ser uma tarefa que consome muito tempo, especialmente se precisar ser repetida em vários locais. Se existir um índice espacial para a camada, a operação será muito mais eficaz.

Pense em uma camada sem um índice espacial como uma lista telefônica na qual os números de telefone não são ordenados ou indexados. A única maneira de encontrar o número de telefone de uma determinada pessoa é ler desde o início até encontrá-lo.

Os índices espaciais não são criados por padrão para uma camada vetorial QGIS, mas você pode criá-los facilmente. Isto é o que você precisa fazer:

- crie índice espacial usando a classe `QgsSpatialIndex()`:

```
index = QgsSpatialIndex()
```

- adicione feições ao índice — o indíce pega o objeto `QgsFeature` e o adiciona à estrutura de dados interna. Você pode criar o objeto manualmente ou usar um de uma chamada anterior para o método `getFeatures()`.

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

- Como alternativa, você pode carregar todas as feições de uma camada ao mesmo tempo usando o carregamento em massa

```
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))
```

6.6 Criando Camadas Vetoriais

Existem várias maneiras de gerar um conjunto de dados de camada vetorial:

- a classe `QgsVectorFileWriter`: Uma classe conveniente para gravar arquivos vetoriais em disco, usando uma chamada estática para `writeAsVectorFormat()` salva a camada vetorial inteira ou cria uma instância da classe e emite chamadas para `addFeature()`. Esta classe suporta todos os formatos vetoriais suportados pelo OGR (GeoPackage, Shapefile, GeoJSON, KML e outros).
- a classe `QgsVectorLayer`: instancia um provedor de dados que interpreta o caminho fornecido (url) da fonte de dados para conectar e acessar os dados. Ele pode ser usado para criar camadas temporárias baseadas em memória (`memory`) e conectar-se a conjuntos de dados OGR (`ogr`), bancos de dados (`postgres`, `spatialite`, `mysql`, `mssql`) e mais (`wfs`, `gpx`, `delimitedtext`...).

6.6.1 De uma instância de `QgsVectorFileWriter`

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

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "ESRI Shapefile"
save_options.fileEncoding = "UTF-8"
transform_context = QgsProject.instance().transformContext()
error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
                                                  "testdata/my_new_shapefile",
```

(continua na próxima página)

(continuação da página anterior)

```
                                                transform_context,
                                                save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
else:
  print(error)
```

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

Você também pode converter campos para torná-los compatíveis com diferentes formatos usando `FieldValueConverter`. Por exemplo, para converter tipos de variáveis de matriz (por exemplo, em Postgres) em um tipo de texto, você pode fazer o seguinte:

```
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 = QgsVectorFileWriter.SaveVectorOptions()
opts.fieldValueConverter = converter
```

Um SRC de destino também pode ser especificado — se uma instância válida de `QgsCoordinateReferenceSystem` for passada como o quarto parâmetro, a camada será transformada nesse SRC.

Para nomes de driver válidos, chame o método `supportedFiltersAndFormats` ou consulte formatos suportados pelo OGR — você deve passar o valor na coluna "Code" como o nome do driver.

Opcionalmente, você pode definir se deseja exportar apenas os recursos selecionados, passar outras opções específicas do driver para criação ou dizer ao gravador para não criar atributos… Há vários outros parâmetros (opcionais); veja a documentação de `QgsVectorFileWriter` para obter detalhes.

6.6.2 Diretamente das feições

```
from qgis.PyQt.QtCore import QVariant

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

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

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

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

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

# add a feature
fet = QgsFeature()

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

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

6.6.3 De uma instância de `QgsVectorLayer`

Entre todos os provedores de dados suportados pela classe `QgsVectorLayer`, vamos nos concentrar nas camadas baseadas em memória. O provedor de memória deve ser usado principalmente por desenvolvedores de complementos ou de 3os. Ele não armazena dados no disco, permitindo que os desenvolvedores os usem como um backend rápido para algumas camadas temporárias.

O provedor suporta campos string, int e double.

O provedor de memória também suporta a indexação espacial, que é ativada chamando a função `createSpatialIndex()`. Depois que o índice espacial for criado, você poderá iterar as feições em regiões menores mais rapidamente (já que não é necessário percorrer todos as feições, apenas aquelas no retângulo especificado).

Um provedor de memória é criado passando `"memory"` como a string do provedor para o construtor `QgsVectorLayer`.

O construtor também usa um URI que define o tipo de geometria da camada, um dos seguintes: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`,``"MultiPolygon"`` ou `"None"`.

O URI também pode especificar o sistema de referência de coordenadas, campos e indexação do provedor de memória no URI. A sintaxe é:

crs=definição Especifica o sistema de referência de coordenadas, onde a definição pode ser qualquer uma das formas aceitas pelo `QgsCoordinateReferenceSystem.createFromString`

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

field=name:type(length,precision) Especifica um atributo da camada. O atributo tem um nome e, opcionalmente, um tipo (número inteiro, duplo ou sequência), comprimento e precisão. Pode haver múltiplas definições de campo.

O exemplo seguinte de URL incorpora todas estas opções

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

O código de exemplo a seguir ilustra a criação e o preenchimento de um provedor de memória

```
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()
```

Finalmente, vamos verificar se tudo correu bem

```
# show some stats
print("fields:", len(pr.fields()))
print("features:", pr.featureCount())
```

(continua na próxima página)

(continuação da página anterior)

```
e = vl.extent()
print("extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum())

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

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

6.7 Aparencia (Simbologia) de Camadas de Vetor

Quando uma camada vetorial está sendo renderizada, a aparência dos dados é dada pelos **renderizador** e **símbolos** associados à camada. Símbolos são classes que cuidam do desenho da representação visual de recursos, enquanto os renderizadores determinam qual símbolo será usado para uma feição específica.

O renderizador para uma determinada camada pode ser obtido como mostrado abaixo:

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

E com essa referência, vamos explorar um pouco

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

```
Type: singleSymbol
```

Existem vários tipos conhecidos de renderizadores disponíveis na biblioteca principal do QGIS:

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

Também pode haver alguns tipos de renderizador personalizados, portanto, nunca assuma que existem apenas esses tipos. Você pode consultar `QgsRendererRegistry` do aplicativo para descobrir os renderizadores disponíveis no momento:

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

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

É possível obter um dump do conteúdo do renderizador em forma de texto - pode ser útil para depuração

```
renderer.dump()
```

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

6.7.1 Renderizador de símbolo único

Você pode obter o símbolo usado para renderização chamando o método meth:*symbol() <qgis.core.QgsSingleSymbolRenderer.symbol>* e alterá-lo com o método `setSymbol()` (observação para desenvolvedores do C++: o renderizador assume a propriedade do símbolo.)

Você pode alterar o símbolo usado por uma camada vetorial específica chamando `setSymbol()` passando uma instância da instância de símbolo apropriada. Símbolos para as camadas de *ponto*, *linha* e *polígono* podem ser criados chamando a função `createSimple()` das classes correspondentes `QgsMarkerSymbol`, `QgsLineSymbol` e `QgsFillSymbol`.

O dicionário passado para `createSimple()` define as propriedades de estilo do símbolo.

Por exemplo, você pode substituir o símbolo usado por uma determinada camada de **ponto** chamando `setSymbol()` passando uma instância de `QgsMarkerSymbol`, como no seguinte exemplo de código:

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

`name` indica a forma do marcador e pode ser um dos seguintes:

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

Para obter a lista completa de propriedades da primeira camada de símbolo de uma instância de símbolo, você pode seguir o código de exemplo:

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

```
{'angle': '0', 'color': '255,0,0,255', 'horizontal_anchor_point': '1', 'joinstyle
↪': 'bevel', 'name': 'square', 'offset': '0,0', 'offset_map_unit_scale': '3x:0,0,
↪0,0,0,0', 'offset_unit': 'MM', 'outline_color': '35,35,35,255', 'outline_style':
↪'solid', 'outline_width': '0', 'outline_width_map_unit_scale': '3x:0,0,0,0,0,0',
↪'outline_width_unit': 'MM', 'scale_method': 'diameter', 'size': '2', 'size_map_
↪unit_scale': '3x:0,0,0,0,0,0', 'size_unit': 'MM', 'vertical_anchor_point': '1'}
```

Isso pode ser útil se você quiser alterar algumas propriedades:

```
# 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:
```

(continua na próxima página)

(continuação da página anterior)

```
props = layer.renderer().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.renderer().setSymbol(QgsMarkerSymbol.createSimple(props))
# show the changes
layer.triggerRepaint()
```

6.7.2 Renderizador de Símbolo Categorizado

Ao usar um renderizador categorizado, é possível consultar e definir o atributo usado para classificação usando `classAttribute()` e `setClassAttribute()`.

Para obter uma lista de categorias

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

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

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

Em que `value()` é o valor usado para discriminação entre categorias, `label()` é um texto usado para a descrição da categoria e `symbol()` retorna o símbolo atribuído.

Geralmente, o renderizador também armazena símbolo e rampa de cores originais que foram usados para a classificação: métodos `sourceColorRamp()` e `sourceSymbol()`.

6.7.3 Renderizador de Símbolo Graduado

Esse renderizador é muito semelhante ao renderizador de símbolo categorizado descrito acima, mas, em vez de um valor de atributo por classe, ele trabalha com intervalos de valores e, portanto, pode ser usado apenas com atributos numéricos.

Para saber mais sobre os intervalos usados no renderizador

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

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

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

você pode novamente usar os métodos `classAttribute` (para encontrar o nome do atributo de classificação), `sourceSymbol` e `sourceColorRamp`. Além disso, existe o método `mode` que determina como os intervalos foram criados: usando intervalos iguais, quantis ou algum outro método.

Se você deseja criar seu próprio renderizador de símbolo graduado, pode fazê-lo como ilustrado no exemplo de trecho abaixo (que cria um arranjo simples de duas classes)

```
from qgis.PyQt import QtGui

myVectorLayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
myTargetField = 'ELEV'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setOpacity(myOpacity)
myRange1 = QgsRendererRange(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbol.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setOpacity(myOpacity)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
↪"EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
```

6.7.4 Trabalhando com Símbolos

Para representação de símbolos, existe a classe base `QgsSymbol` com três classes derivadas:

- `QgsMarkerSymbol` — para feições de pontos
- `QgsLineSymbol` — para feições de linha
- `QgsFillSymbol` — para feições de polígono

Todo símbolo consiste em uma ou mais camadas de símbolos (classes derivadas de `QgsSymbolLayer`). As camadas de símbolo fazem a renderização real, a própria classe de símbolo serve apenas como um contêiner para as camadas de símbolo.

Tendo uma instância de um símbolo (por exemplo, de um renderizador), é possível explorá-lo: o método `type` diz se é um marcador, linha ou símbolo com preenchimento. Existe um método `dump` que retorna uma breve descrição do símbolo. Para obter uma lista de camadas de símbolos:

```
marker_symbol = QgsMarkerSymbol()
for i in range(marker_symbol.symbolLayerCount()):
```

(continua na próxima página)

(continuação da página anterior)

```
    lyr = marker_symbol.symbolLayer(i)
    print("{}: {}".format(i, lyr.layerType()))
```

```
0: SimpleMarker
```

Para descobrir a cor do símbolo, use o método `color` e `setColor` para alterar sua cor. Além disso, com os símbolos dos marcadores, é possível consultar o tamanho e a rotação do símbolo com os métodos `size` e `angle`. Para símbolos de linha, o método `width` retorna a largura da linha.

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

Trabalhando com Camadas de Símbolos

Como dito anteriormente, as camadas de símbolos (subclasses de `QgsSymbolLayer`) determinam a aparência das feições. Existem várias classes básicas de camadas de símbolos para uso geral. É possível implementar novos tipos de camadas de símbolos e, assim, personalizar arbitrariamente como as feição serão renderizadas. O método `layerType()` identifica exclusivamente a classe da camada de símbolo — os básicos e o padrão são `SimpleMarker`, `SimpleLine` e `SimpleFill` tipos de camadas de símbolos.

Você pode obter uma lista completa dos tipos de camadas de símbolos que pode criar para uma determinada classe de camadas de símbolos com o seguinte código:

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

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

A classe `QgsSymbolLayerRegistry` gerencia um banco de dados de todos os tipos de camada de símbolo disponíveis.

Para acessar os dados da camada de símbolo, use o método `properties()` que retorna um dicionário de valores-chave de propriedades que determinam a aparência. Cada tipo de camada de símbolo possui um conjunto específico de propriedades que ele usa. Além disso, existem os métodos genéricos `color`, `size`, `angle` e `width`, com suas definições correspondentes. É claro que o tamanho e o ângulo estão disponíveis apenas para as camadas de símbolos dos marcadores e a largura para as camadas de símbolos de linha.

Criando Tipos de Camadas de Símbolos Personalizadas

Imagine que você gostaria de personalizar a maneira como os dados são renderizados. Você pode criar sua própria classe de camada de símbolo que desenhará as feições exatamente como você deseja. Aqui está um exemplo de um marcador que desenha círculos vermelhos com raio especificado

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

class FooSymbolLayer(QgsMarkerSymbolLayer):

  def __init__(self, radius=4.0):
```

(continua na próxima página)

(continuação da página anterior)

```
        QgsMarkerSymbolLayer.__init__(self)
        self.radius = radius
        self.color = QColor(255,0,0)

    def layerType(self):
       return "FooMarker"

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

    def startRender(self, context):
      pass

    def stopRender(self, context):
        pass

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

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

O método `layerType` determina o nome da camada de símbolo; deve ser único entre todas as camadas de símbolos. O método `properties` é usado para persistência de atributos. O método `clone` deve retornar uma cópia da camada de símbolo com todos os atributos exatamente iguais. Finalmente, existem métodos de renderização: `startRender` é chamado antes de renderizar o primeiro recurso, `stopRender` quando a renderização é concluída, e `renderPoint` é chamado para fazer a renderização. As coordenadas do(s) ponto(s) já estão transformadas nas coordenadas de saída.

Para polilinhas e polígonos, a única diferença seria no método de renderização: você usaria `renderPolyline` que recebe uma lista de linhas, enquanto :meth:`renderPolygon <qgis.core. QgsFillSymbolLayer.renderPolygon> `recebe uma lista de pontos no anel externo como primeiro parâmetro e uma lista de anéis internos (ou None) como segundo parâmetro.

Geralmente, é conveniente adicionar uma GUI para definir atributos do tipo de camada de símbolo para permitir que os usuários personalizem a aparência: no caso do nosso exemplo acima, podemos permitir que o usuário defina o raio do círculo. O código a seguir implementa esse widget

```
from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

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

    def setSymbolLayer(self, layer):
```

(continua na próxima página)

(continuação da página anterior)

```
        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()"))
```

Este widget pode ser incorporado na caixa de diálogo de propriedades do símbolo. Quando o tipo de camada de símbolo é selecionado na caixa de diálogo de propriedades do símbolo, ele cria uma instância da camada de símbolos e uma instância do widget da camada de símbolos. Em seguida, chama o método `setSymbolLayer` para atribuir a camada de símbolo ao widget. Nesse método, o widget deve atualizar a interface do usuário para refletir os atributos da camada de símbolo. O método `symbolLayer` é usado para recuperar a camada de símbolo novamente pela caixa de diálogo de propriedades para usá-la para o símbolo.

Em cada alteração de atributos, o widget deve emitir o sinal `changed()` para permitir que o diálogo de propriedades atualize a visualização do símbolo.

Agora falta apenas a cola final: fazer o QGIS entender essas novas classes. Isso é feito adicionando a camada de símbolo ao registro. É possível usar a camada de símbolo também sem adicioná-la ao registro, mas algumas funcionalidades não funcionarão: por exemplo carregamento de arquivos de projeto com as camadas de símbolos personalizados ou incapacidade de editar os atributos da camada na GUI.

Você terá que criar metadados para a camada de símbolos

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

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

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

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

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

Você deve passar o tipo de camada (igual ao retornado pela camada) e o tipo de símbolo (marcador/linha/preenchimento) para o construtor da classe pai. O método `createSymbolLayer()` cuida da criação de uma instância da camada de símbolos com atributos especificados no dicionário *props*. E existe o método `createSymbolLayerWidget()` que retorna o widget de configurações para esse tipo de camada de símbolo.

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

6.7.5 Criando Renderizadores Personalizados

Pode ser útil criar uma nova implementação de renderizador se você desejar personalizar as regras de como selecionar símbolos para renderização de recursos. Alguns casos de uso em que você deseja fazer isso: o símbolo é determinado a partir de uma combinação de campos, o tamanho dos símbolos muda dependendo da escala atual etc.

O código a seguir mostra um renderizador personalizado simples que cria dois símbolos de marcador e escolhe aleatoriamente um deles para cada feição

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


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

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

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

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

  def usedAttributes(self, context):
    return []

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

O construtor da classe pai `QgsFeatureRenderer` precisa de um nome de renderizador (que deve ser exclusivo entre os renderizadores). O método `symbolForFeature` é o método que decide qual símbolo será usado para um recurso específico. `startRender` e `stopRender` cuida da inicialização/finalização da renderização de símbolo. O método `usedAttributes` pode retornar uma lista de nomes de campos que o representante espera estar presente. Finalmente, o método `clone` deve retornar uma cópia do renderizador.

Como nas camadas de símbolos, é possível anexar uma GUI para a configuração do renderizador. Ele deve ser derivado de `QgsRendererWidget`. O código de amostra a seguir cria um botão que permite ao usuário definir o primeiro símbolo

```
from qgis.gui import QgsRendererWidget, QgsColorButton


class RandomRendererWidget(QgsRendererWidget):
  def __init__(self, layer, style, renderer):
    super().__init__(layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButton()
```

(continua na próxima página)

(continuação da página anterior)

```
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.btn1.colorChanged.connect(self.setColor1)

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

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

O construtor recebe instâncias da camada ativa (`QgsVectorLayer`), do estilo global (`QgsStyle`) e do renderizador atual. Se não houver renderizador ou o renderizador tiver um tipo diferente, ele será substituído pelo nosso novo renderizador; caso contrário, usaremos o renderizador atual (que já possui o tipo que precisamos). O conteúdo do widget deve ser atualizado para mostrar o estado atual do renderizador. Quando a caixa de diálogo do renderizador é aceita, o método do widget `renderer` é chamado para obter o renderizador atual - ele será atribuído à camada.

A última parte ausente são os metadados e a inclusão do renderizador no registro, caso contrário, o carregamento de camadas com o renderizador não funcionará e o usuário não poderá selecioná-lo na lista de renderizadores. Vamos terminar o nosso exemplo RandomRenderer

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

class RandomRendererMetadata(QgsRendererAbstractMetadata):

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

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

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

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

Da mesma forma que nas camadas de símbolos, o construtor de metadados abstratos aguarda o nome do renderizador, nome visível para os usuários e, opcionalmente, o nome do ícone do renderizador. O método `createRenderer` passa uma instância de `QDomElement` que pode ser usada para restaurar o estado do renderizador da árvore DOM. O método `createRendererWidget` cria o widget de configuração. Ele não precisa estar presente ou pode retornar `None` se o renderizador não vier com a GUI.

Para associar um ícone ao renderizador, você pode atribuí-lo no construtor `QgsRendererAbstractMetadata` como um terceiro argumento (opcional) — o construtor da classe base na função RandomRendererMetadata `__init__`` () se torna

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

O ícone também pode ser associado posteriormente, usando o método `setIcon` da classe de metadados. O ícone pode ser carregado de um arquivo (como mostrado acima) ou de um recurso Qt (PyQt5 inclui o compilador .qrc para Python).

6.8 Outros Tópicos

TODO:

- criando/modificando símbolos
- trabalhando com estilo (`QgsStyle`)
- trabalhando com rampa de cores (`QgsColorRamp`)
- explorando a camada de símbolos e os registros do renderizador

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

CAPÍTULO 7

Manipulação Geométrica

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

Pontos, linhas e polígonos que representam um feição espacial são comumente referidos como geometrias. No QGIS, eles são representados com a classe `QgsGeometry`.

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.

A descrição e as especificações de todas as possíveis construções e relações de geometrias estão disponíveis no OGC Simple Feature Access Standards para detalhes mais avançados.

7.1 Construção de Geométria

O PyQGIS fornece várias opções para criar uma geometria:

- 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)
```

As coordinadas são dadas usando a classe `QgsPoint` ou a classe `QgsPointXY`. A diferençae entre estas classes é que a classe `QgsPoint` suporta M e Z dimensões.

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

Um polígono é representado por uma lista de anéis lineares (ou seja, fformas de linhas fechadas). O primeiro anel é o anel externo (limite); os anéis subsequentes opcionais são orifícios no polígono. Observe que, diferentemente de alguns programas, o QGIS fechará o anel para você, não sendo necessário duplicar o primeiro ponto como o último.

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())
```

7.2 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())
  # output: 2 for LineString
if gPolygon.wkbType() == QgsWkbTypes.Polygon:
  print(gPolygon.wkbType())
  # output: 3 for Polygon
```

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

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

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

```
Point
LineString
Polygon
```

There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

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

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

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()`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

```
from qgis.core import (
    QgsCoordinateReferenceSystem,
```

(continua na próxima página)

(continuação da página anterior)

```
    QgsCoordinateTransform,
    QgsProject,
    QgsPointXY,
)
```

CAPÍTULO 8

Suporte a projeções

8.1 Sistemas de Referencia de Coordenadas

Os sistemas de referência de coordenadas (SRC) são encapsulados pela classe `QgsCoordinateReferenceSystem`. Instâncias desta classe podem ser criadas de várias maneiras diferentes:

- especificar CRS pela sua identificação

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

  O QGIS suporta identificadores SRC diferentes com os seguintes formatos:

  - `EPSG:<code>` — ID ID atribuído pela organização EPSG - manipulado com `createFromOgcWms()`
  - `POSTGIS:<srid>`— ID usado em bancos de dados PostGIS databases - manipulados com `createFromSrid()`
  - `INTERNAL:<srsid>` — ID usados no banco de dados interno do QGIS - manipulados com `createFromSrsId()`
  - `PROJ:<proj>` - manipulado com `createFromProj()`
  - `WKT:<wkt>` - manipulado com `createFromWkt()`

Se nenhum prefixo for especificado, assume-se a definição WKT.

- especifica SRC pelo seu texto conhecido (WKT)

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

- cria um SRC inválido e usa uma das funções `create*` para inicializá-lo. No exemplo a seguir, usamos uma string Proj para inicializar a projeção.

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

É interessante verificar se a criação (ou seja, pesquisa no banco de dados) do SRC foi bem-sucedida: `isValid()` deve retornar `True`.

Observe que, para a inicialização dos sistemas de referência espacial, o QGIS precisa procurar valores apropriados em seu banco de dados interno `srs.db`. Portanto, no caso de você criar um aplicativo independente, precisará definir os caminhos corretamente com `QgsApplication.setPrefixPath ()`, caso contrário, ele vai falhar quando tentar encontrar o banco de dados. Se você estiver executando os comandos no console do QGIS Python ou desenvolvendo um complemento, tudo já está configurado para você.

Acesso à informação do sistema de referência espacial:

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

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

Saída:

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

8.2 Transformação de SRC

Você pode fazer a transformação entre diferentes sistemas de referência espacial usando a classe `QgsCoordinateTransform`. A maneira mais fácil de usá-lo é criar um SRC de origem e destino e construir uma instância `QgsCoordinateTransform` com eles e o projeto atual. Em seguida, basta chamar repetidamente a função `transform ()` para fazer a transformação. Por padrão, ele faz a transformação, mas é capaz de fazer também a transformação inversa.

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

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

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

Saída:

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

from qgis.PyQt.QtCore import Qt, QRectF

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

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

CAPÍTULO 9

Usando a Tela do Mapa

- *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*

O widget de tela Mapa é provavelmente o widget mais importante no QGIS, pois mostra o mapa composto de camadas de mapas sobrepostas e permite a interação com o mapa e as camadas. A tela sempre mostra uma parte do mapa definida pela extensão atual da tela. A interação é feita através do uso de **ferramentas de mapa**: existem ferramentas para panorâmica, zoom, identificação de camadas, medição, edição de vetores e outras. Semelhante a outros programas gráficos, sempre há uma ferramenta ativa e o usuário pode alternar entre as ferramentas disponíveis.

A tela do mapa é implementada com a classe `QgsMapCanvas` no módulo `qgis.gui`. A implementação é baseada na estrutura Qt Graphics View. Essa estrutura geralmente fornece uma superfície e uma exibição em que itens gráficos personalizados são colocados e o usuário pode interagir com eles. Assumiremos que você está familiarizado o suficiente com o Qt para entender os conceitos da cena, visualização e itens gráficos. Caso contrário, leia a visão geral do framework.

Sempre que o mapa for panorâmico, tem o zoom aumentado/diminuído (ou alguma outra ação que desencadeia uma atualização), o mapa é renderizado novamente na extensão atual. As camadas são renderizadas em uma imagem (usando a classe `QgsMapRendererJob`) e essa imagem é exibida na tela. A classe `QgsMapCanvas` classe também controla a atualização do mapa renderizado. Além deste item que atua como plano de fundo, pode haver mais **itens de tela de mapa**.

Os itens típicos da tela de mapa são rubber bands (usados para medição, edição de vetor etc.) ou marcadores de vértice. Os itens de tela geralmente são usados para fornecer feedback visual para as ferramentas de mapa; por exemplo, ao criar um novo polígono, a ferramenta de mapa cria um item de tela rubber band que mostra a forma atual do polígono. Todos os itens da tela de mapa são subclasses de `QgsMapCanvasItem`, que adiciona mais funcionalidade aos objetos básicos `QGraphicsItem`.

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

- tela do mapa — para visualização do mapa
- itens da tela de mapa — itens adicionais que podem ser mostrados na tela do mapa

- ferramentas de mapa — para interação com a tela do mapa

9.1 Incorporar o Mapa da Tela

A tela de mapa é um widget como qualquer outro widget Qt, portanto, usá-lo é tão simples quanto criar e mostrar.

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

Isso produz uma janela independente com tela de mapa. Também pode ser incorporado a um widget ou janela existente. Ao usar arquivos `.ui` e o Qt Designer, coloque um `QWidget` no formulário e promova-o para uma nova classe: defiina` *QgsMapCanvas*` como nome da classe e defina `qgis.gui` como arquivo de cabeçalho. O `pyuic5` cuidará dele. Essa é uma maneira muito conveniente de incorporar a tela. A outra possibilidade é escrever manualmente o código para construir a tela do mapa e outros widgets (como filhos de uma janela ou caixa de diálogo principal) e criar um 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)
```

(Caso você esteja se perguntando, `Qt` vem do módulo `PyQt.QtCore` e `Qt.white` é uma das instâncias predefinidas do `QColor`).

Agora é hora de adicionar algumas camadas do mapa. Primeiro abriremos uma camada e a adicionaremos ao projeto atual. Em seguida, definiremos a extensão da tela e a lista de camadas para a tela.

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

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

9.2 Bandas raster e fazedor de vértices

Para mostrar alguns dados adicionais na parte superior do mapa na tela, use os itens da tela do mapa. É possível criar classes de item de tela personalizadas (abordadas abaixo), no entanto, existem duas classes úteis de itens de tela por conveniência: `QgsRubberBand` para desenhar polilinhas ou polígonos e `QgsVertexMarker` para pontos de desenho. Os dois trabalham com coordenadas do mapa, para que a forma seja movida/redimensionada automaticamente quando a tela for panorâmica ou ampliada.

Para mostrar uma polilinha:

```
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)
```

Os itens da tela são vinculados à cena da tela. Para ocultá-los temporariamente (e mostrá-los novamente), use as combinações `hide()` e `show()`. Para remover completamente o item, você deve removê-lo da cena da tela

```
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 também pode ser usado para desenhar pontos, mas a classe `QgsVertexMarker` é mais adequada para isso (`QgsRubberBand` apenas desenha um retângulo ao redor do ponto desejado).

Você pode usar o marcador de vértice assim:

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

Isto irá desenhar uma cruz vermelha na posição **[10,45]**. É possível personalizar o tipo de ícone, tamanho, cor e largura da caneta

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

Para ocultar temporariamente os marcadores de vértice e removê-los da tela, use os mesmos métodos usados para rubber bands.

9.3 Usando Ferramentas de Mapas na Tela

O exemplo a seguir constrói uma janela que contém uma tela de mapa e ferramentas básicas de mapa para panorâmica e zoom do mapa. As ações são criadas para a ativação de cada ferramenta: as panôramicas são feitas com `QgsMapToolPan`, aumentando/diminuindo o zoom com algumas instâncias `QgsMapToolZoom`. As ações são definidas como verificáveis e, posteriormente, atribuídas às ferramentas para permitir o manuseio automático do estado marcado/desmarcado das ações - quando uma ferramenta de mapa é ativada, sua ação é marcada como selecionada e a ação da ferramenta de mapa anterior é desmarcada. As ferramentas de mapa são ativadas usando o método `setMapTool()`.

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

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

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

(continua na próxima página)

(continuação da página anterior)

```
        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)
```

Você pode tentar o código acima no editor de console do Python. Para chamar a janela da tela, adicione as seguintes linhas para instanciar a classe `MyWnd`. Eles renderizarão a camada atualmente selecionada na tela recém-criada

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

9.4 Desenhar ferramenta de mapa personalizada

Você pode escrever suas ferramentas personalizadas, para implementar um comportamento personalizado nas ações executadas pelos usuários na tela.

As ferramentas de mapa devem herdar de `QgsMapTool`, classe ou qualquer classe derivada, e selecionadas como ferramentas ativas na tela usando o `setMapTool ()` como já vimos.

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.

```
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)
```

(continua na próxima página)

(continuação da página anterior)

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

9.5 Desenhar itens da tela do mapa

Aqui está um exemplo de um item de tela personalizado que desenha um círculo:

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

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

  def center(self):
    return self.center

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

  def size(self):
    return self.size

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

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


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

Os trechos de código nesta página precisam das seguintes importações:

```
import os

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

(continua na próxima página)

(continuação da página anterior)

```
    QgsLayoutItem,
    QgsLayoutPoint,
    QgsLayoutSize,
    QgsUnitTypes,
    QgsProject,
    QgsFillSymbol,
)

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

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

CAPÍTULO 10

Renderização em impressão de mapas

- *Renderização simples*
- *Renderizando camadas com CRS diferente*
- *Saída usando layout de impressão*
  - *Exportando o layout*
  - *Exportando um atlas como layout*

Geralmente, existem duas abordagens em que os dados de entrada devem ser renderizados como um mapa: seja rápido usando *QgsMapRendererJob* ou produza uma saída mais ajustada compondo o mapa com a classe `QgsLayout`.

10.1 Renderização simples

A renderização é feita criando um objeto `QgsMapSettings` para definir as configurações de renderização e, em seguida, construindo um `QgsMapRendererJob` com essas configurações. O último é usado para criar a imagem resultante.

He aquí un ejemplo:

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

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

render = QgsMapRendererParallelJob(settings)

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

(continua na próxima página)

(continuação da página anterior)

```
16
17 render.finished.connect(finished)
18
19 render.start()
```

10.2 Renderizando camadas com CRS diferente

Se você tiver mais de uma camada e eles tiverem SRC diferente, o exemplo simples acima provavelmente não funcionará: para obter os valores corretos a partir dos cálculos de extensão, você deve definir explicitamente o SRC de destino

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

10.3 Saída usando layout de impressão

O layout de impressão é uma ferramenta muito útil se você deseja obter uma saída mais sofisticada do que a simples renderização mostrada acima. É possível criar layouts de mapas complexos que consistem em visualizações de mapas, rõtulos, legendas, tabelas e outros elementos que geralmente estão presentes nos mapas em papel. Os layouts podem ser exportados para PDF, imagens raster ou diretamente impressas em uma impressora.

O layout consiste de várias classes. Todos eles pertencem à biblioteca principal. O aplicativo QGIS possui uma GUI conveniente para a colocação dos elementos, embora não esteja disponível na biblioteca da GUI. Se você não estiver familiarizado com a estrutura Qt Graphics View, recomendamos que verifique a documentação agora, porque o layout é baseado nela .

A classe central do layout é a classe `QgsLayout`, que é derivada da classe Qt QGraphicsScene. Vamos criar uma instância dele:

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

Agora podemos adicionar vários elementos (mapa, rõtulo, …) ao layout. Todos esses objetos são representados por classes que herdam da classe base `QgsLayoutItem`.

Aqui está uma descrição de alguns dos principais itens de layout que podem ser adicionados a um 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
- forma básica
- forma baseada em nós

```
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

Depois que um item é adicionado ao layout, ele pode ser movido e redimensionado:

```
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)
```

Além de criar os itens de layout manualmente, o QGIS oferece suporte a modelos de layout que são essencialmente composições com todos os itens salvos em um arquivo .qpt (com sintaxe XML).

Quando a composição estiver pronta (os itens de layout foram criados e adicionados à composição), podemos continuar produzindo uma saída raster e/ou vetorial.

10.3.1 Exportando o layout

Para exportar um layout, a classe :class: *QgsLayoutExporter <qgis.core.QgsLayoutExporter>* deve ser usada.

```
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 `exportToImage ()` caso deseje exportar para uma imagem em vez de um arquivo PDF.

10.3.2 Exportando um atlas como layout

Se você deseja exportar todas as páginas de um layout que tenha a opção atlas configurada e ativada, use o método `atlas ()` no exportador (:class:` QgsLayoutExporter <qgis.core.QgsLayoutExporter>`) com pequenos ajustes. No exemplo a seguir, as páginas são exportadas para imagens PNG:

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

Observe que as saídas serão salvas na pasta do caminho base, usando a expressão do nome do arquivo de saída configurada no atlas.

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

CAPÍTULO 11

Expressões, filtragem e cálculo dos valores

- *Expressões de Análise*
- *Expressões de Avaliação*
  - *Expressões Básicas*
  - *Expressões com características*
  - *Filtrando uma camada com expressões*
- *Manipulando erros de expressão*

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`

11.1 Expressões de Análise

O exemplo a seguir mostra como verificar se uma determinada expressão pode ser analisada corretamente:

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

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

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

11.2 Expressões de Avaliação

Expressões podem ser usadas em diferentes contextos, por exemplo, para filtrar recursos ou calcular novos valores de campo. De qualquer forma, a expressão deve ser avaliada. Isso significa que seu valor é calculado executando as etapas computacionais especificadas, que podem variar de aritmética simples a expressões agregadas.

11.2.1 Expressões Básicas

Essa expressão básica é avaliada como 1, o que significa que é verdadeiro:

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

11.2.2 Expressões com características

Para avaliar uma expressão em relação a um recurso, um objeto `QgsExpressionContext` deve ser criado e passado à função de avaliação para permitir que a expressão acesse os valores de campo do recurso.

O exemplo a seguir mostra como criar um recurso com um campo chamado "Coluna" e como adicionar esse recurso ao contexto da expressão.

```
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)
```

A seguir, é apresentado um exemplo mais completo de como usar expressões no contexto de uma camada vetorial, para calcular novos valores de campo:

```
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
```

(continua na próxima página)

(continuação da página anterior)

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

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

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

```
876.5
```

11.2.3 Filtrando uma camada com expressões

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)
```

11.3 Manipulando erros de expressão

Erros relacionados à expressão podem ocorrer durante a análise ou análise de expressão:

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

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

CAPÍTULO 12

Leitura e Armazenamento de Configurações

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.

Podemos diferenciar entre vários tipos de configurações:

- **configurações globais —** estão vinculadas ao usuário em uma máquina específica. O próprio QGIS armazena muitas configurações globais, por exemplo, tamanho da janela principal ou tolerância de snap padrão. As configurações são tratadas usando a classe `QgsSettings`, por exemplo, através dos métodos `setValue()` e `value()`.

  Aqui você pode ver um exemplo de como esses métodos são usados.

```
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)
```

  O segundo parâmetro do método `value ()` é opcional e especifica o valor padrão retornado se não houver um valor anterior definido para o nome da configuração transmitida.

  Para um método para pré-configurar os valores padrão das configurações globais através do arquivo `global_settings.ini`, veja deploying_organization para mais detalhes.

- **configurações do projeto —** variam entre projetos diferentes e, portanto, estão conectados a um arquivo de projeto. A cor de fundo da tela de mapa ou o sistema de referência de coordenadas de destino (SRC)

são exemplos — fundo branco e WGS84 podem ser adequados para um projeto, enquanto fundo amarelo e projeção UTM são melhores para outro.

Um exemplo de uso a seguir.

```
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)
```

Como você pode ver, o método `writeEntry()` é usado para todos os tipos de dados, mas existem vários métodos para ler o valor de configuração novamente, e o correspondente deve ser selecionado para cada tipo de dados.

- **configurações da camada de mapa** — essas configurações estão relacionadas a uma instância específica de uma camada de mapa com um projeto. Eles *não* estão conectados à fonte de dados subjacente de uma camada; portanto, se você criar duas instâncias da camada de mapa de um shapefile, elas não compartilharão as configurações. As configurações são armazenadas dentro do arquivo do projeto; portanto, se o usuário abrir o projeto novamente, as configurações relacionadas à camada estarão lá novamente. O valor para uma determinada configuração é recuperado usando o método `customProperty()` e pode ser definido usando o método `setCustomProperty()`.

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

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

from qgis.gui import (
    QgsMessageBar,
)

from qgis.PyQt.QtWidgets import (
```

(continua na próxima página)

(continuação da página anterior)

```
    QSizePolicy,
    QPushButton,
    QDialog,
    QGridLayout,
    QDialogButtonBox,
)
```

CAPÍTULO 13

Comunicação com o usuário

- *Mostrando mensagens. A classe QgsMessageBar*
- *Mostrando progresso*
- *Carregando*
  - *QgsMessageLog*
  - *The python built in logging module*

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.

13.1 Mostrando mensagens. A classe QgsMessageBar

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)
```

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

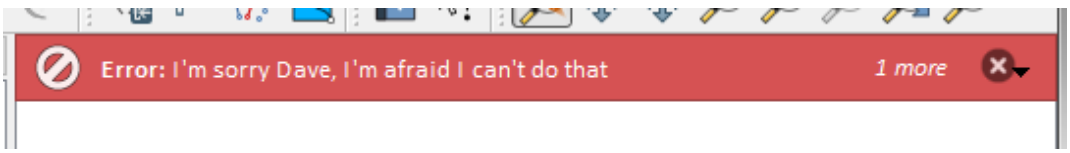


Fig. 13.1: Barra de mensagem do QGIS

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)
```

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

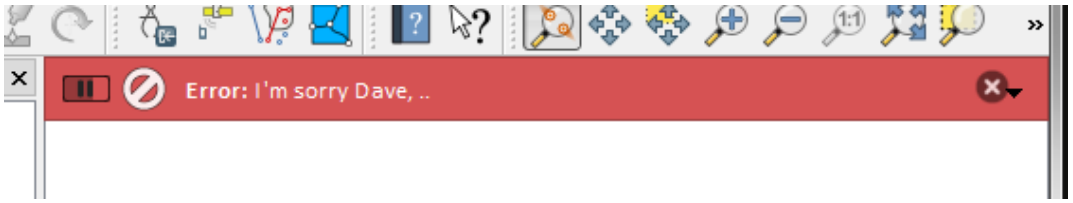


Fig. 13.2: Barra de Mensagem QGIS como temporizar

Os exemplos acima mostram uma barra de erro, mas o parâmetro `` level`` pode ser usado para criar mensagens de aviso ou informações, usando enumeração `Qgis.MessageLevel`. Você pode usar até 4 níveis diferentes

0. Info
1. Aviso
2. Crítico
3. Sucesso

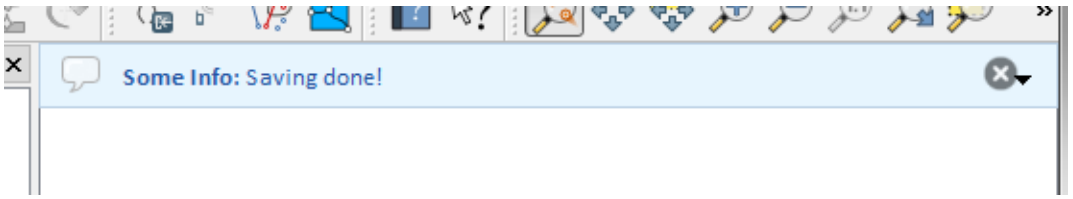


Fig. 13.3: Barra de mensagens QGIS (informação)

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

```
def showError():
    pass

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

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

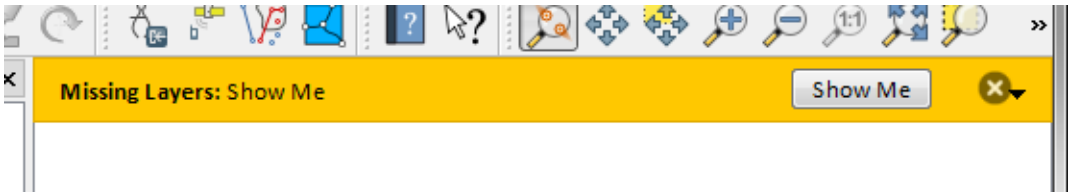


Fig. 13.4: Barra de mensagens QGIS com um botão

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

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

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

Fig. 13.5: Barra de mensagens QGIS em diálogo personalizado

13.2 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()
```

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

Além disso, você pode usar a barra de status interna para relatar o progresso, como no próximo exemplo:

```
vlayer = iface.activeLayer()

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

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

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

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

13.3 Carregando

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

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

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

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

13.3.1 QgsMessageLog

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

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

Nota: Você pode ver a saída da `QgsMessageLog` no log_message_panel

13.3.2 The python built in logging module

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

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

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

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

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

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

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

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

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

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

from qgis.PyQt.QtNetwork import QSslCertificate
```

CAPÍTULO 14

Infraestrutura de autenticação

- *Introdução*
- *Glossário*
- *QgsAuthManager o ponto de entrada*
  - *Inicie o gerente e defina a master password*
  - *Preencha o authdb com uma nova entrada de Configuração de Autenticação*
    - *Métodos de Autenticação disponíveis*
    - *Preencher Autoridades*
    - *Gerenciar pacotes configuráveis de PKI com QgsPkiBundle*
  - *Remover uma entrada de authdb*
  - *Leave authcfg expansion to QgsAuthManager*
    - *Exemplos PKI com outros provedores de dados*
- *Adapte complementos para usar a infraestrutura de autenticação*
- *GUIs de autenticação*
  - *GUI para selecionar credenciais*
  - *GUI do Editor de Autenticação*
  - *GUI do Editor de Autoridades*

14.1 Introdução

A referência do usuário da infraestrutura de autenticação pode ser lida no Manual do Usuário no parágrafo authentication_overview.

Este capítulo descreve as práticas recomendadas para usar o sistema de autenticação da perspectiva do desenvolvedor.

O sistema de autenticação é amplamente utilizado no QGIS Desktop pelos provedores de dados sempre que são necessárias credenciais para acessar um recurso específico, por exemplo, quando uma camada estabelece uma conexão com um banco de dados Postgres.

Existem também alguns widgets na biblioteca QGIS que os desenvolvedores de complementos podem usar para integrar facilmente a infraestrutura de autenticação ao seu código:

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

Uma boa referência de código pode ser lida na infraestrutura de autenticação tests code.

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

14.2 Glossário

Aqui estão algumas definições dos objetos mais comuns tratados neste capítulo.

Master Password Senha para permitir acesso e descriptografia de credenciais armazenadas no Banco de Dados de Autenticação QGIS

Banco de Dados de Autenticação O *Master Password* banco de dados criptografado sqlite `qgis-auth.db` onde *Authentication Configuration* são armazenados, como por exemplo, usuário/senha, certificados e chaves pessoais, Autoridades de Certificação

Banco de Dados de Autenticação *Authentication Database*

Configuração de Autenticação Um conjunto de dados de autenticação, dependendo de *Authentication Method*, como por exemplo, o Método de autenticação básica armazena o par de usuário/senha.

Configurar Autenticação *Authentication Configuration*

Método de Autenticação Um método específico usado para se autenticar. Cada método possui seu próprio protocolo usado para obter o nível autenticado. Cada método é implementado como uma biblioteca compartilhada carregada dinamicamente durante o init da infraestrutura de autenticação QGIS.

14.3 QgsAuthManager o ponto de entrada

O singleton `QgsAuthManager` é o ponto de entrada para usar as credenciais armazenadas no QGIS *Authentication DB* criptografado, i.e. o arquivo `qgis-auth.db` sob a pasta ativa user profile.

Essa classe cuida da interação do usuário: pedindo para definir uma master password ou usando-a de forma transparente para acessar informações armazenadas criptografadas.

14.3.1 Inicie o gerente e defina a master password

O trecho a seguir fornece um exemplo para definir a master password para abrir o acesso às configurações de autenticação. Os comentários do código são importantes para entender o trecho.

```
authMgr = QgsApplication.authManager()

# check if QgsAuthManager has already been initialized... a side effect
# of the QgsAuthManager.init() is that AuthDbPath is set.
# QgsAuthManager.init() is executed during QGIS application init and hence
# you do not normally need to call it directly.
if authMgr.authenticationDatabasePath():
    # already initilised => we are inside a QGIS app.
    if authMgr.masterPasswordIsSet():
        msg = 'Authentication master password not recognized'
        assert authMgr.masterPasswordSame("your master password"), msg
    else:
        msg = 'Master password could not be set'
        # The verify parameter check if the hash of the password was
        # already saved in the authentication db
        assert authMgr.setMasterPassword("your master password",
                                         verify=True), msg
else:
    # outside qgis, e.g. in a testing environment => setup env var before
    # db init
    os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
    msg = 'Master password could not be set'
    assert authMgr.setMasterPassword("your master password", True), msg
    authMgr.init("/path/where/located/qgis-auth.db")
```

14.3.2 Preencha o authdb com uma nova entrada de Configuração de Autenticação

Qualquer credencial armazenada é uma instância de *Authentication Configuration* da classe `QgsAuthMethodConfig` acessada usando uma string exclusiva como a que segue:

```
authcfg = 'fm1s770'
```

essa string é gerada automaticamente ao criar uma entrada usando a API ou a GUI do QGIS, mas pode ser útil configurá-la manualmente para um valor conhecido caso a configuração precise ser compartilhada (com credenciais diferentes) entre vários usuários em uma organização.

`QgsAuthMethodConfig` é a classe base para qualquer *Authentication Method*. Qualquer Método de Autenticação define um mapa de hash de configuração onde as informações de autenticação serão armazenadas. A seguir, um trecho útil para armazenar credenciais do caminho PKI para um usuário hipotético alice:

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

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

(continua na próxima página)

(continuação da página anterior)

```
newAuthCfgId = config.id()
assert newAuthCfgId
```

Métodos de Autenticação disponíveis

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

1. `Basic` Autenticação de usuário e senha
2. `Esri-Token` ESRI token based authentication
3. `Identity-Cert` Autenticação de certificado de identidade
4. Autenticação de OAuth2 `OAuth2`
5. `PKI-Paths` autenticação de caminhos PKI
6. `PKI-PKCS#12` PKI PKCS#12 autenticação

Preencher Autoridades

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

Gerenciar pacotes configuráveis de PKI com QgsPkiBundle

Uma classe de conveniência para empacotar pacotes configuráveis PKI compostos na cadeia SslCert, SslKey e CA é a classe `QgsPkiBundle`. A seguir, um trecho para obter uma senha protegida:

```
# add alice cert in case of key with pwd
caBundlesList = []  # List of CA bundles
bundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                    "/path/to/alice-key_w-pass.pem",
                                    "unlock_pwd",
                                    caBundlesList )
assert bundle is not None
# You can check bundle validity by calling:
# bundle.isValid()
```

Consulte a documentação da classe `QgsPkiBundle` para extrair cert/key/CAs do pacote.

14.3.3 Remover uma entrada de authdb

Podemos remover uma entrada de *Authentication Database* usando seu identificador `authcfg` com o seguinte trecho:

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

14.3.4 Leave authcfg expansion to QgsAuthManager

A melhor maneira de usar um *Authentication Config* armazenado em *Authentication DB* é referenciá-lo com o identificador exclusivo `authcfg`. Expandir significa convertê-lo de um identificador para um conjunto completo de credenciais. A melhor prática para usar os *Authentication Config*s armazenados, é deixá-los gerenciados automaticamente pelo gerenciador de autenticação. O uso comum de uma configuração armazenada é conectar-se a um serviço habilitado para autenticação, como um WMS ou WFS ou a uma conexão de banco de dados.

Nota: Leve em consideração que nem todos os provedores de dados QGIS estão integrados à infraestrutura de autenticação. Cada método de autenticação, derivado da classe base `QgsAuthMethod` e suporta um conjunto diferente de Provedores. Por exemplo, o método `certIdentity()` suporta a seguinte lista de provedores:

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

Saída de amostra:

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

Por exemplo, para acessar um serviço WMS usando credenciais armazenadas identificadas com `authcfg = 'fm1s770'`, basta usar o `authcfg` no URL da fonte de dados, como no seguinte trecho:

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

Em maiúsculas, o provedor `wms` cuidará para expandir o parâmetro URI `authcfg` com credencial antes de definir a conexão HTTP.

Aviso: O desenvolvedor precisaria deixar a expansão `authcfg` para `QgsAuthManager`, dessa forma ele garantirá que a expansão não seja feita muito cedo.

Geralmente, uma string de URI, criada usando a classe `QgsDataSourceURI`, é usada para definir uma fonte de dados da seguinte maneira:

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

Nota: O parâmetro `False` é importante para evitar a expansão completa do URI do ID `authcfg` presente no URI.

Exemplos PKI com outros provedores de dados

Outro exemplo pode ser lido diretamente nos testes QGIS anteriores, como em test_authmanager_pki_ows ou test_authmanager_pki_postgres.

14.4 Adapte complementos para usar a infraestrutura de autenticação

Muitos complementos de terceiros estão usando o enableplib2 ou outras bibliotecas de rede Python para gerenciar conexões HTTP em vez de integrar-se com `QgsNetworkAccessManager` e sua integração relacionada à Infraestrutura de autenticação.

Para facilitar essa integração, uma função auxiliar Python foi criada chamada `NetworkAccessManager`. Seu código pode ser encontrado aqui.

Essa classe auxiliar pode ser usada como no seguinte trecho:

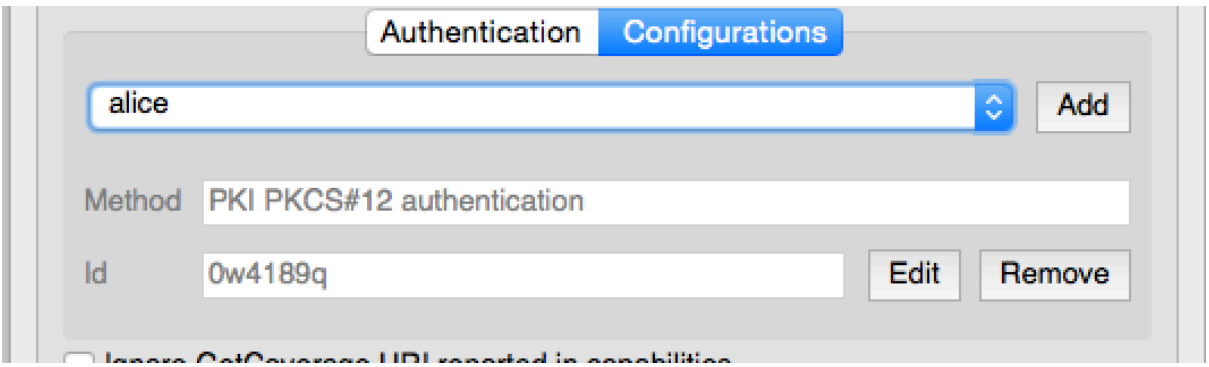
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 GUIs de autenticação

Neste parágrafo, estão listadas as GUIs disponíveis úteis para integrar a infraestrutura de autenticação em interfaces personalizadas.

14.5.1 GUI para selecionar credenciais

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`.



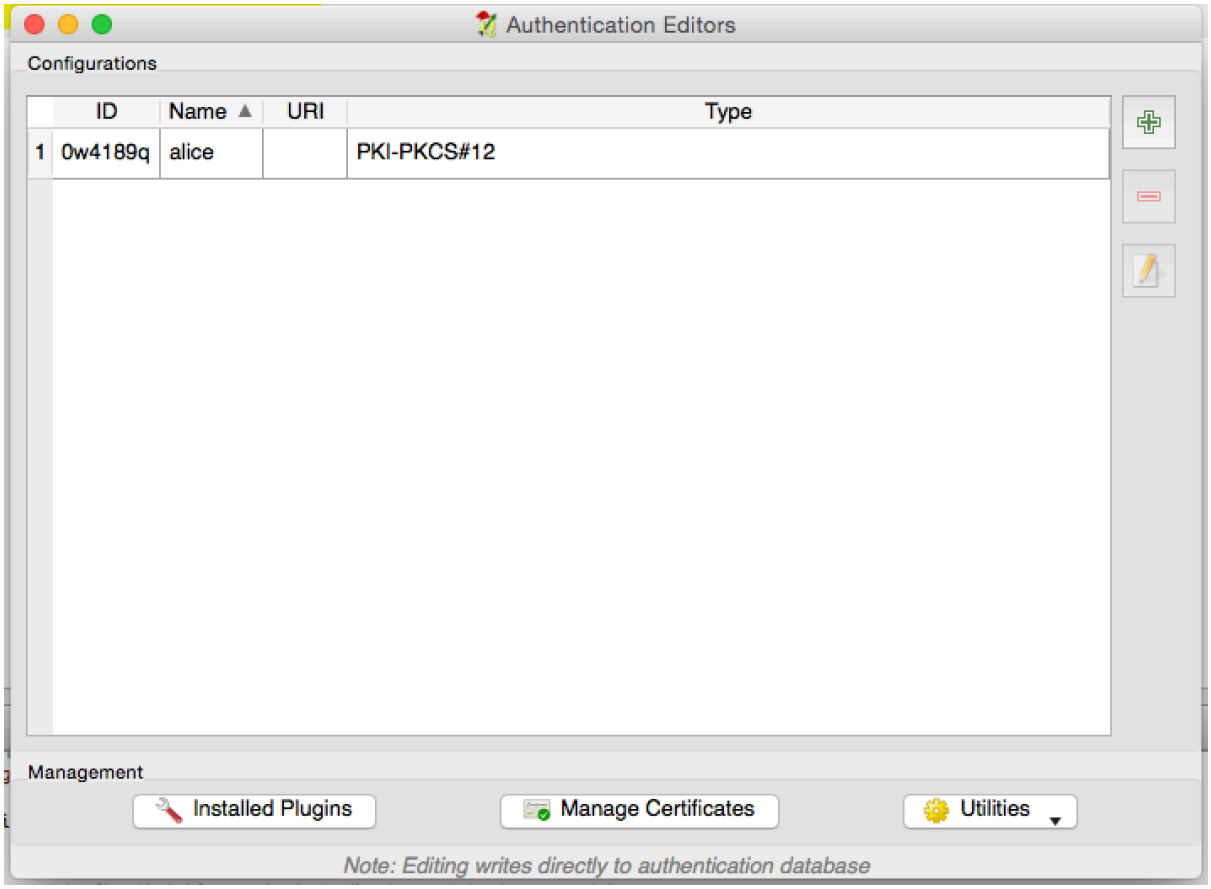
e pode ser usado como no seguinte trecho:

```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with `parent`
parent = QWidget()  # Your GUI parent widget
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
# GUI has to be integrated
tabGui = QTabWidget()
tabGui.insertTab( 1, gui, "Configurations" )
```

O exemplo acima é retirado da fonte do QGIS code. O segundo parâmetro do construtor da GUI refere-se ao tipo de provedor de dados. O parâmetro é usado para restringir os *Authentication Method*s compatíveis com o provedor especificado.

14.5.2 GUI do Editor de Autenticação

A GUI completa usada para gerenciar credenciais, autoridades e acessar os utilitários de autenticação é gerenciada pela classe `QgsAuthEditorWidgets`.



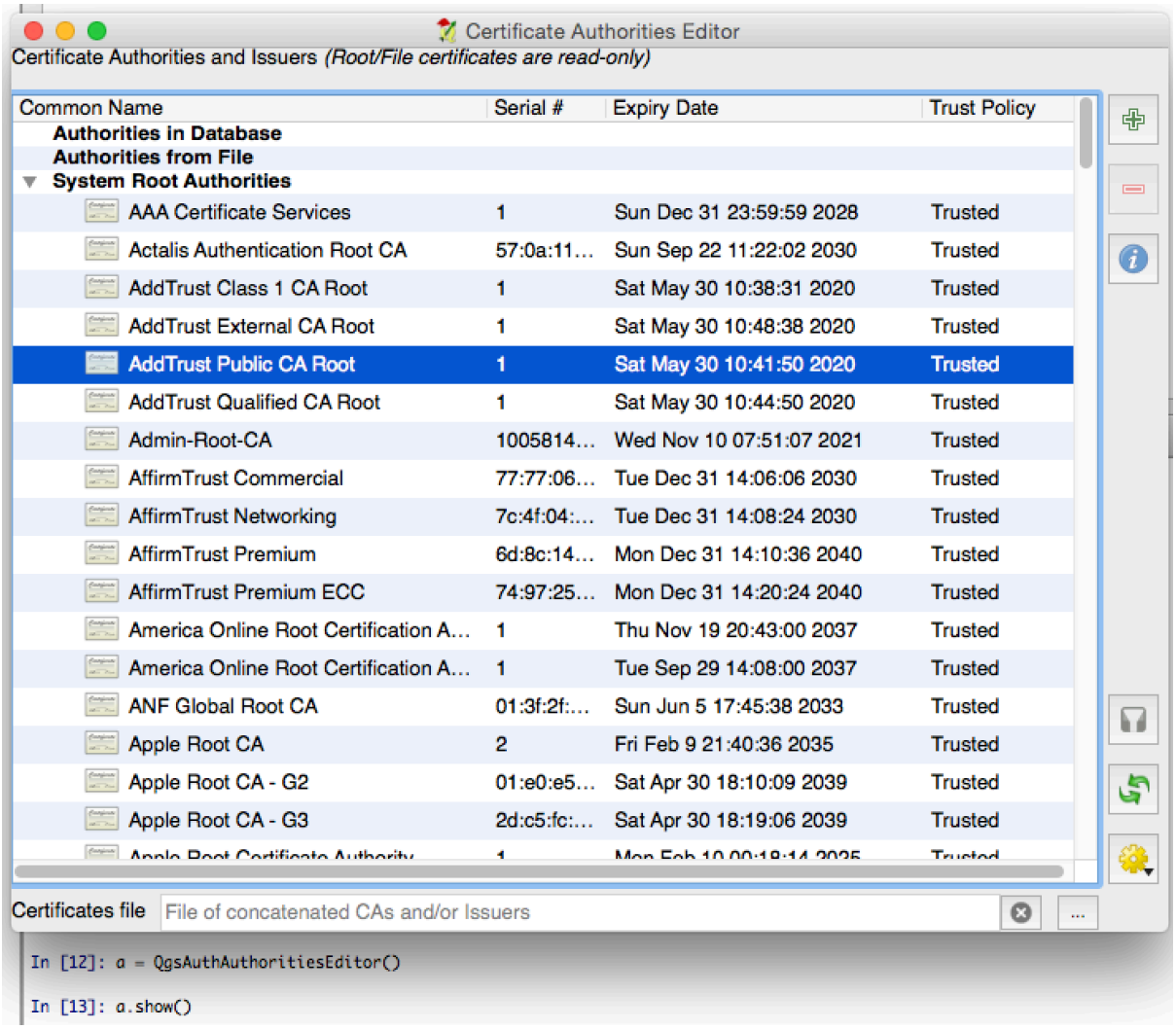
e pode ser usado como no seguinte trecho:

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

An integrated example can be found in the related test.

14.5.3 GUI do Editor de Autoridades

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



e pode ser usado como no seguinte trecho:

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

CAPÍTULO 15

Tarefas - trabalho pesado em segundo plano

15.1 Introdução

O processamento em segundo plano usando threads é uma maneira de manter uma interface de usuário responsdendo quando o processamento pesado está em andamento. As tarefas podem ser usadas para obter threading no QGIS.

Uma tarefa (`QgsTask`) é um contêiner para o código a ser executado em segundo plano, e o gerenciador de tarefas (`QgsTaskManager`) é usado para controlar a execução das tarefas. Essas classes simplificam o processamento em segundo plano no QGIS, fornecendo mecanismos para sinalização, relatórios de progresso e acesso ao status dos processos em segundo plano. As tarefas podem ser agrupadas usando subtarefas.

O gerenciador de tarefas global (encontrado com `QgsApplication.taskManager()`) é normalmente usado. Isso significa que suas tarefas podem não ser as únicas controladas pelo gerenciador de tarefas.

Existem várias maneiras de criar uma tarefa QGIS:

- Crie sua própria tarefa estendendo `QgsTask`

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

- Criar uma tarefa a partir de uma função

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

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

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

- Criar uma tarefa a partir de um algoritmo de processamento

```
params = dict()
context = QgsProcessingContext()
feedback = QgsProcessingFeedback()
```

(continua na próxima página)

(continuação da página anterior)

```
4
5  buffer_alg = QgsApplication.instance().processingRegistry().algorithmById(
   ↪'native:buffer')
6  task = QgsProcessingAlgRunnerTask(buffer_alg, params, context,
7                                    feedback)
```

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

Dependências entre tarefas podem ser descritas usando a função `addSubTask` de `QgsTask`. Quando uma dependência é declarada, o gerenciador de tarefas determina automaticamente como essas dependências serão executadas. Sempre que possível, as dependências serão executadas em paralelo para satisfazê-las o mais rápido possível. Se uma tarefa da qual outra tarefa depende for cancelada, a tarefa dependente também será cancelada. Dependências circulares podem tornar possíveis impasses, portanto, tenha cuidado.

Se uma tarefa depende da disponibilidade de uma camada, isso pode ser indicado usando a função `setDependentLayers` de `QgsTask`. Se uma camada da qual uma tarefa depende não estiver disponível, a tarefa será cancelada.

Após a criação da tarefa, ela pode ser agendada para execução usando a função `addTask` do gerenciador de tarefas. A adição de uma tarefa ao gerente transfere automaticamente a propriedade dessa tarefa para o gerente, e o gerente limpa e exclui as tarefas após a execução. O agendamento das tarefas é influenciado pela prioridade da tarefa, que é definida em `addTask`.

O status das tarefas pode ser monitorado usando `QgsTask` e sinais e funções de `QgsTaskManager`.

15.2 Exemplos

15.2.1 Estendendo QgsTask

Neste exemplo, `RandomIntegerSumTask` estende `QgsTask` e gerará 100 números inteiros aleatórios entre 0 e 500 durante um período especificado. Se o número aleatório for 42, a tarefa será abortada e uma exceção será gerada. Várias instâncias de `RandomIntegerSumTask` (com subtarefas) são geradas e adicionadas ao gerenciador de tarefas, demonstrando dois tipos de dependências.

```
1  import random
2  from time import sleep
3
4  from qgis.core import (
5      QgsApplication, QgsTask, QgsMessageLog,
6      )
7
8  MESSAGE_CATEGORY = 'RandomIntegerSumTask'
9
10 class RandomIntegerSumTask(QgsTask):
11     """This shows how to subclass QgsTask"""
12
13     def __init__(self, description, duration):
14         super().__init__(description, QgsTask.CanCancel)
15         self.duration = duration
16         self.total = 0
17         self.iterations = 0
18         self.exception = None
```

(continua na próxima página)

(continuação da página anterior)

```
    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
        Raising exceptions will crash QGIS, so we handle them
        internally and raise them in self.finished
        """
        QgsMessageLog.logMessage('Started task "{}"'.format(
                                     self.description()),
                                 MESSAGE_CATEGORY, Qgis.Info)
        wait_time = self.duration / 100
        for i in range(100):
            sleep(wait_time)
            # use setProgress to report progress
            self.setProgress(i)
            arandominteger = random.randint(0, 500)
            self.total += arandominteger
            self.iterations += 1
            # check isCanceled() to handle cancellation
            if self.isCanceled():
                return False
            # simulate exceptions to show how to abort task
            if arandominteger == 42:
                # DO NOT raise Exception('bad value!')
                # this would crash QGIS
                self.exception = Exception('bad value!')
                return False
        return True

    def finished(self, result):
        """
        This function is automatically called when the task has
        completed (successfully or not).
        You implement finished() to do whatever follow-up stuff
        should happen after the task is complete.
        finished is always called from the main thread, so it's safe
        to do GUI operations and raise Python exceptions here.
        result is the return value from self.run.
        """
        if result:
            QgsMessageLog.logMessage(
                'RandomTask "{name}" completed\n' \
                'RandomTotal: {total} (with {iterations} '\
              'iterations)'.format(
                    name=self.description(),
                    total=self.total,
                    iterations=self.iterations),
                MESSAGE_CATEGORY, Qgis.Success)
        else:
            if self.exception is None:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" not successful but without '\
                    'exception (probably the task was manually '\
                    'canceled by the user)'.format(
                        name=self.description()),
                    MESSAGE_CATEGORY, Qgis.Warning)
            else:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" Exception: {exception}'.format(
```

(continua na próxima página)

(continuação da página anterior)

```
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception

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


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

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

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

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

15.2.2 Tarefa da função

Crie uma tarefa a partir de uma função (`doSomething` neste exemplo). O primeiro parâmetro da função conterá `QgsTask` para a função. Um parâmetro importante (nomeado) é `on_finished`, que especifica uma função que será chamada quando a tarefa for concluída. A função `doSomething` neste exemplo possui um parâmetro adicional com nome `wait_time`.

```
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

```

(continua na próxima página)

(continuação da página anterior)

```
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

# Create a few tasks
```

(continua na próxima página)

(continuação da página anterior)

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

```
RandomIntegerSumTask(0): Started task "waste cpu subtask short"
RandomTaskFromFunction(0): Started task Waste cpu 1
RandomTaskFromFunction(0): Started task Waste cpu 2
RandomTaskFromFunction(0): Task Waste cpu 2 completed
RandomTotal: 23263 ( with 100 iterations)
RandomTaskFromFunction(0): Task Waste cpu 1 completed
RandomTotal: 25044 ( with 100 iterations)
```

15.2.3 Tarefa de um algoritmo de processamento

Crie uma tarefa que use o algoritmo qgis:randompointsinextent para gerar 50000 pontos aleatórios dentro de uma extensão especificada. O resultado é adicionado ao projeto de forma segura.

```
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)
```

Veja também: https://opengis.ch/2018/06/22/threads-in-pyqgis3/.

CAPÍTULO 16

Desenvolvendo complementos Python

16.1 Estruturando Complementos Python

- *Escrevendo um complemento*
  - *Arquivos de complementos*
- *Conteúdo do complemento*
  - *Metadados do complemento*
  - *__init__.py*
  - *mainPlugin.py*
  - *Arquivo de Recurso*
- *Documentação*
- *Tradução*
  - *Requisitos de software*
  - *Arquivos e diretório*
    - *Arquivo .pro*
    - *Arquivo .ts*
    - *Arquivo .qm*
  - *Traduzindo usando Makefile*
  - *Carregue o complemento*
- *Dicas e truques*
  - *Recarregador de Complementos*
  - *Acessando Complementos*
  - *Mensagens de log*
  - *Compartilhe seu complemento*

Para criar um complemento, aqui estão alguns passos a seguir:

1. *Idéia*: tenha uma idéia sobre o que você deseja fazer com seu novo complemento QGIS. Por que você quer fazer isso? Que problema você quer resolver? Já existe outro complemento para esse problema?
2. *Criar arquivos*: alguns são essenciais (veja *Arquivos de complementos*)
3. *Escrevendo o código*: Escreva o código nos arquivos apropriados
4. *Teste*: *Reload your plugin* para verificar se tudo está OK
5. *Publicando*: publique seu complemento no repositório QGIS ou faça seu próprio repositório como um "arsenal" de "armas GIS" pessoais.

16.1.1 Escrevendo um complemento

Desde a introdução dos complementos Python no QGIS, vários complementos apareceram. A equipe do QGIS mantém um *Repositório Oficial de complementos Python*. Você pode usar sua fonte para aprender mais sobre programação com PyQGIS ou descobrir se você está duplicando o esforço de desenvolvimento.

Arquivos de complementos

Aqui está a estrutura de diretórios do nosso exemplo de complemento

```
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*
```

Qual é o significado dos arquivos:

- `__init__.py` = O ponto de partida do complemento. Ele deve ter o método `classFactory()` e pode ter qualquer outro código de inicialização.
- `mainPlugin.py` = O código principal de trabalho do complemento. Contém todas as informações sobre as ações do complemento e o código principal.
- `resources.qrc` = O documento .xml criado pelo Qt Designer. Contém caminhos relativos aos recursos dos formulários.
- `resources.py` = A tradução do arquivo .qrc descrito acima para Python.
- `form.ui` = A GUI criada pelo Qt Designer.
- `form.py` = A tradução do form.ui descrito acima para Python.
- `metadata.txt` = Contém informações gerais, versão, nome e alguns outros metadados usados pelo site de complementos e pela infraestrutura de complementos.

Aqui é uma maneira de criar os arquivos básicos (esqueleto) de um complemento típico do QGIS Python.

Existe um complemento do QGIS chamado Plugin Builder 3 que cria um modelo de complemento para o QGIS. Esta é a opção recomendada, pois produz fontes compatíveis com 3.x.

Aviso: Se você planeja fazer o upload do complemento para *Repositório Oficial de complementos Python*, verifique se o seu complemento segue algumas regras adicionais necessárias para o complemento *Validação*

16.1.2 Conteúdo do complemento

Aqui você pode encontrar informações e exemplos sobre o que adicionar em cada um dos arquivos na estrutura de arquivos descrita acima.

Metadados do complemento

Primeiro, o gerenciador de complementos precisa recuperar algumas informações básicas sobre o complemento, como nome, descrição etc. O arquivo `metadata.txt` é o lugar certo para colocar essas informações.

Nota: Todos os metadados devem estar na codificação UTF-8.

Nome dos metadados	Requerido	Notas
nome	True	uma string curta que contém o nome do complemento
qgisMinimumVersion	True	notação dotted da versão mínima do QGIS
qgisMaximumVersion	False	notação dotted da versão máxima do QGIS
descrição	True	texto breve que descreve o complemento, HTML não permitido
sobre	True	texto mais longo que descreve o complemento em detalhes, não é permitido HTML
versão	True	string curta com a notação dotted da versão
autor	True	nome do autor
email	True	email do autor, mostrado apenas no site para usuários logados, mas visível no Gerenciador de complementos após a instalação do complemento
changelog	False	string, pode ser multilinha, não é permitido HTML
experimental	False	flag booleano, *True* ou` False` - *True* se esta versão for experimental
descontinuada	False	O sinalizador booleano, *True* ou` False`, aplica-se a todo o complemento e não apenas à versão carregada
etiquetas	False	lista separada por vírgula, são permitidos espaços dentro de tetiquetas individuais
página inicial	False	um URL válido apontando para a página inicial do seu complemento
repositório	True	uma URL válida para o repositório de código-fonte
tracker	False	um URL válido para tickets e relatórios de erros
ícone	False	um nome de arquivo ou um caminho relativo (relativo à pasta base do pacote compactado do complemento) de uma imagem amigável da Web (PNG, JPEG)
categoria	False	uma de *Raster*, *Vetor*, *Banco de dados* e *Web*
dependencies_de_complemento	False	Lista separada por vírgula do tipo PIP de outros complementos para instalar
servidor	False	boolean flag, *True* or *False*, determines if the plugin has a server interface
hasProcessingProvider	False	flag booleano, *True* ou` False`, determina se o complemento fornece algoritmos de processamento

Por padrão, os complementos são colocados no menu *Complementos* (veremos na próxima seção como adicionar uma entrada de menu ao seu complemento), mas também podem ser colocados nos menus *Raster*, *Vetor*, *Banco de dados* e *Web*.

Existe uma entrada de metadados de "categoria" correspondente para especificar isso, para que o complemento possa ser classificado de acordo. Essa entrada de metadados é usada como dica para os usuários e informa onde (em qual menu) o complemento pode ser encontrado. Os valores permitidos para "categoria" são: Vetor, Raster, Banco de dados ou Web. Por exemplo, se o seu complemento estará disponível no menu *Raster*, adicione-o a `metadata.txt`

```
category=Raster
```

Nota: Se *qgisMaximumVersion* estiver vazio, ele será automaticamente definido para a versão principal mais` .99` quando carregado no *Repositório Oficial de complementos Python*.

Um exemplo para este metadata.txt

```
; the next section is mandatory

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

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

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

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

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

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

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

; Since QGIS 3.8, a comma separated list of plugins to be installed
; (or upgraded) can be specified.
; The example below will try to install (or upgrade) "MyOtherPlugin" version 1.12
; and any version of "YetAnotherPlugin"
plugin_dependencies=MyOtherPlugin==1.12,YetAnotherPlugin
```

__init__.py

Este arquivo é requerido pelo sistema de importação do Python. Além disso, o QGIS exige que este arquivo contenha uma função `classFactory()`, chamada quando o complemento é carregado no QGIS. Ele recebe uma referência à `QgisInterface` e deve retornar um objeto da classe do seu complemento a partir de `mainplugin.py` — no nosso caso, é chamado `TestPlugin` (veja abaixo). É assim que `__init__.py` deve se parecer

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

# any other initialisation needed
```

mainPlugin.py

É aqui que a mágica acontece e é assim que a mágica se parece: (por exemplo `mainPlugin.py`)

```
from qgis.PyQt.QtGui import *
from qgis.PyQt.QtWidgets import *

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

class TestPlugin:

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

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

    # add toolbar button and menu item
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

    # connect to signal renderComplete which is emitted when canvas
    # rendering is done
    self.iface.mapCanvas().renderComplete.connect(self.renderTest)

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

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

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

  def renderTest(self, painter):
```

(continua na próxima página)

(continuação da página anterior)

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

As únicas funções de complemento que devem existir no arquivo principal de complemento (por exemplo `mainPlugin.py`) são:

- `__init__`, que dá acesso à interface QGIS
- `initGui ()` chamado quando o complemento é carregado
- `unload ()` chamado quando o complemento é descarregado

No exemplo acima, é usado `addPluginToMenu`. Isso adicionará a ação do menu correspondente ao menu *Complementos*. Existem métodos alternativos para adicionar a ação a um menu diferente. Aqui está uma lista desses métodos:

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

Todos eles têm a mesma sintaxe do método `addPluginToMenu`.

É recomendável adicionar o menu do complemento a um desses métodos predefinidos para manter a consistência na organização das entradas do complemento. No entanto, você pode adicionar seu grupo de menus personalizado diretamente à barra de menus, como o próximo exemplo demonstra:

```
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()
```

Não se esqueça de definir `QAction` e `QMenu` `objectName` com um nome específico ao seu complemento para que ele possa ser personalizado.

Arquivo de Recurso

Você pode ver que em `initGui ()` usamos um ícone do arquivo de recursos (chamado `resources.qrc` no nosso caso)

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

É bom usar um prefixo que não colide com outros plugins ou qualquer parte do QGIS; caso contrário, você poderá obter recursos que não deseja. Agora você só precisa gerar um arquivo Python que conterá os recursos. É feito com **`pyrcc5`** command:

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

Nota: Em ambientes Windows, a tentativa de executar o comando **`pyrcc5`** no prompt de comando ou no Powershell provavelmente resultará no erro "O Windows não pode acessar o dispositivo, caminho ou arquivo especificado [...]". A solução mais fácil é provavelmente usar o OSGeo4W Shell, mas se você estiver confortável modificando a variável de ambiente PATH ou especificando o caminho do executável explicitamente, poderá encontrá-lo em `<Your QGIS Install Directory>\bin\pyrcc5.exe`.

E isso é tudo … nada complicado :)

Se você tiver feito tudo corretamente, poderá encontrar e carregar seu complemento no gerenciador de complementos e ver uma mensagem no console quando o ícone da barra de ferramentas ou o item de menu apropriado estiver selecionado.

Ao trabalhar em um complemento real, é aconselhável gravá-lo em outro diretório (ativo) e criar um makefile que gere arquivos de recursos + interface do usuário e instale o complemento na instalação do QGIS.

16.1.3 Documentação

A documentação para o complemento pode ser escrita como arquivos de ajuda em HTML. O módulo `qgis.utils` fornece uma função `showPluginHelp()`, que abrirá o navegador de arquivos de ajuda, da mesma maneira que outras ajudas do QGIS.

A função `showPluginHelp()` procura por arquivos de ajuda no mesmo diretório que o módulo de chamada. Ele procurará, por sua vez, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` `e :file:`index.html`, exibindo o que encontrar primeiro. Aqui `ll_cc` é o local QGIS. Isso permite que várias traduções da documentação sejam incluídas no complemento.

A função `showPluginHelp()` também pode usar os parâmetros packageName, que identifica um complemento específico para o qual a ajuda será mostrada, o nome do arquivo, que pode substituir o "índice" nos nomes dos arquivos pesquisados e a seção, que é o nome de uma marca de âncora html no documento em que o navegador será posicionado.

16.1.4 Tradução

Com algumas etapas, você pode configurar o ambiente para a localização do complemento para que, dependendo das configurações de localidade do seu computador, o complemento seja carregado em diferentes idiomas.

Requisitos de software

A maneira mais fácil de criar e gerenciar todos os arquivos de tradução é instalar o Qt Linguist. Em um ambiente GNU/Linux baseado no Debian, você pode instalá-lo digitando:

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

Arquivos e diretório

Ao criar o complemento, você encontrará a pasta `i18n` no diretório principal do complemento.

Todos os arquivos de tradução devem estar dentro deste diretório.

Arquivo .pro

Primeiro, você deve criar um arquivo `.pro`, que é um arquivo *projeto* que pode ser gerenciado pelo **Qt Linguist**.

Neste arquivo `.pro`, você deve especificar todos os arquivos e formulários que deseja traduzir. Este arquivo é usado para configurar os arquivos e variáveis de localização. Um possível arquivo de projeto, correspondendo à estrutura de nosso *example plugin*:

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

Seu complemento pode seguir uma estrutura mais complexa e pode ser distribuído por vários arquivos. Se for esse o caso, lembre-se de que `pylupdate5`, o programa que usamos para ler o arquivo `.pro` e atualizar a string traduzível, não expande caracteres curinga, portanto, você deve colocar todos os arquivos explicitamente no arquivo `.pro`. O arquivo do seu projeto pode se parecer com algo assim:

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

Além disso, o arquivo `your_plugin.py` é o arquivo que *chama* todos os menus e submenus do seu complemento na barra de ferramentas QGIS e você deseja traduzir todos eles.

Finalmente, com a variável *TRANSLATIONS*, você pode especificar os idiomas de tradução que deseja.

Aviso: Certifique-se de nomear o arquivo `ts` como `your_plugin_` + `language` + `.ts`, caso contrário, o carregamento do idioma falhará! Use o atalho de 2 letras para o idioma (**it** para italiano, **de** para alemão, etc…)

Arquivo .ts

Depois de criar o `.pro`, você estará pronto para gerar os arquivos `.ts` para o(s) idioma(s) do seu complemento.

Abra um terminal, vá para o diretório `your_plugin/i18n` e digite:

```
pylupdate5 your_plugin.pro
```

você deve ver os arquivos `your_plugin_language.ts`.

Abra o arquivo `.ts` com **Qt Linguist** e comece a traduzir.

Arquivo .qm

Quando você terminar de traduzir seu complemento (se algumas strings de caracteres não forem concluídas, o idioma de origem será usado), você deverá criar o arquivo `.qm` (o arquivo `.ts` compilado que será usado por QGIS).

Basta abrir um cd do terminal no diretório `your_plugin/i18n` e digite:

```
lrelease your_plugin.ts
```

Agora, no diretório `i18n`, você verá o(s) arquivo(s) `your_plugin.qm`.

Traduzindo usando Makefile

Como alternativa, você pode usar o makefile para extrair mensagens do código python e das caixas de diálogo Qt, se você criou seu complemento com o Plugin Builder. No início do Makefile, há uma variável LOCALES:

```
LOCALES = en
```

Adicione a abreviação do idioma a essa variável, por exemplo, para o idioma húngaro:

```
LOCALES = en hu
```

Agora você pode gerar ou atualizar o arquivo `hu.ts` (e também o arquivo `en.ts`) a partir das fontes:

```
make transup
```

Depois disso, você atualizou o arquivo `.ts` para todos os idiomas definidos na variável LOCALES. Use **Qt Linguist** para traduzir as mensagens do programa. Terminando a tradução, os arquivos `.qm` podem ser criados pelo transcompile:

```
make transcompile
```

Você precisa distribuir arquivos `.ts` com o seu complemento.

Carregue o complemento

Para ver a tradução do seu complemento, abra o QGIS, altere o idioma (*Configurações -> Opções -> Geral*) e reinicie o QGIS.

Você deve ver seu complemento no idioma correto.

> **Aviso:** Se você alterar algo no seu complemento (novas UIs, novo menu, etc.), será necessário **gerar novamente** a versão de atualização do arquivo `.ts` e `.qm`, portanto, execute novamente o comando acima.

16.1.5 Dicas e truques

Recarregador de Complementos

Durante o desenvolvimento do seu complemento, você frequentemente precisará recarregá-lo no QGIS para teste. Isso é muito fácil usando o complemento **Plugin Reloader**. Você pode encontrá-lo com o Plugin Manager.

Acessando Complementos

Você pode acessar todas as classes de complementos instalados no QGIS usando python, o que pode ser útil para fins de depuração.

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

Mensagens de log

Os complementos têm sua própria guia dentro de log_message_panel.

Compartilhe seu complemento

O QGIS está hospedando centenas de complementos no repositório de complementos. Considere compartilhar o seu! Isso ampliará as possibilidades do QGIS e as pessoas poderão aprender com seu código. Todos os complementos hospedados podem ser encontrados e instalados no QGIS com o Gerenciador de Complementos.

Informações e requisitos aqui: plugins.qgis.org.

16.2 Partes de código

- *Como chamar um método por um atalho de teclas*
- *Como alternar Camadas*
- *Como acessar a tabela de atributos das feições selecionadas*
- *Interface for plugin in the options dialog*

Esta seção apresenta trechos de código para facilitar o desenvolvimento de complementos.

16.2.1 Como chamar um método por um atalho de teclas

No complemento, adicione à `initGui()`

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

A `unload()` adicione

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

O método chamado quando CTRL+I é pressionado

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

16.2.2 Como alternar Camadas

Há uma API para acessar camadas na legenda. Aqui está um exemplo que alterna a visibilidade da camada ativa

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

16.2.3 Como acessar a tabela de atributos das feições selecionadas

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

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

# The method requires one parameter (the new value for the second
# field of the selected feature(s)) and can be called by
change_value(50)
```

16.2.4 Interface for plugin in the options dialog

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

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

```
class MyPluginOptionsFactory(QgsOptionsWidgetFactory):

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

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

```

(continua na próxima página)

(continuação da página anterior)

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


class ConfigOptionsPage(QgsOptionsPageWidget):

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

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

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


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

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

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


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

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

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

16.3 Usando Camadas de Complementos

Se o seu complemento usa seus próprios métodos para renderizar uma camada de mapa, escrever seu próprio tipo de camada com base em `QgsPluginLayer` pode ser a melhor maneira de implementar isso.

16.3.1 Subclassificação QgsPluginLayer

Abaixo está um exemplo de uma implementação QgsPluginLayer mínima. É baseado no código original do complemento de exemplo de Marca de Água.

O renderizador personalizado é a parte do implemento que define o desenho real na tela.

```
class WatermarkLayerRenderer(QgsMapLayerRenderer):

    def __init__(self, layerId, rendererContext):
        super().__init__(layerId, rendererContext)

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

class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

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

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

    def setTransformContext(self, ct):
        pass

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

    def readXml(self, node, context):
        pass

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

A camada de complemento pode ser adicionada ao projeto e à tela como qualquer outra camada de mapa:

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

Ao carregar um projeto que contém essa camada, é necessária uma classe de fábrica:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

```

(continua na próxima página)

(continuação da página anterior)

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

    # You can also add GUI code for displaying custom information
    # in the layer properties
    def showLayerProperties(self, layer):
        pass


# Keep a reference to the instance in Python so it won't
# be garbage collected
plt =  WatermarkPluginLayerType()

assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)
```

16.4 Configurações de IDE para gravar e depurar complementos

- *Complementos úteis para escrever complementos Python*
- *Uma observação sobre como configurar seu IDE no Linux e Windows*
- *Depurando usando o Pyscripter IDE (Windows)*
- *Depurando usando Eclipse e PyDev*
  - *Instalação*
  - *Configurando o Eclipse*
  - *Configurando o depurador*
  - *Fazendo o eclipse entender a API*
- *Depurando com PyCharm no Ubuntu com um QGIS compilado*
- *Debugging using PDB*

Embora cada programador tenha seu editor preferido de IDE/Texto, aqui estão algumas recomendações para configurar IDE populares para escrever e depurar complementos QGIS Python.

16.4.1 Complementos úteis para escrever complementos Python

Alguns complementos são convenientes ao escrever complementos Python. De *Complmentos → Gerenciar e instalar complementos...*, instale:

- *Recarregador de complementos*: permite recarregar um complemento e obter novas alterações sem reiniciar o QGIS.
- *Primeiros Socorros*: Isso adicionará um console Python e um depurador local para inspecionar variáveis quando uma exceção é gerada a partir de umcomplemento.

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.*

16.4.2 Uma observação sobre como configurar seu IDE no Linux e Windows

No Linux, tudo o que geralmente precisa ser feito é adicionar os locais da biblioteca QGIS à variável de ambiente `PYTHONPATH` do usuário. Na maioria das distribuições, isso pode ser feito editando file:*~/.bashrc* ou `~/.bash-profile` com a seguinte linha (testada no OpenSUSE Tumbleweed):

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

Salve o arquivo e implemente as configurações do ambiente usando o seguinte comando shell:

```
source ~/.bashrc
```

No Windows, você precisa ter as mesmas configurações de ambiente e usar as mesmas bibliotecas e intérpretes do QGIS. A maneira mais rápida de fazer isso é modificar o arquivo de lote de inicialização do QGIS.

Se você usou o Instalador do OSGeo4W, poderá encontrá-lo na pasta `bin` da sua instalação do OSGeo4W. Procure algo como `C:\OSGeo4W\bin\qgis-unstable.bat`.

16.4.3 Depurando usando o Pyscripter IDE (Windows)

Para usar o Pyscripter IDE, aqui está o que você deve fazer:

1. Faça uma cópia de `qgis-unstable.bat` e renomeie-o como `pyscripter.bat`.
2. Abra-o em um editor. E remova a última linha, a que inicia o QGIS.
3. Adicione uma linha que aponte para o seu executável Pyscripter e adicione o argumento da linha de comando que define a versão do Python a ser usada
4. Adicione também o argumento que aponta para a pasta em que o Pyscripter pode encontrar a dll Python usada pelo QGIS; você pode encontrá-lo na pasta bin da instalação do OSGeoW

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

5. Agora, quando você clicar duas vezes nesse arquivo em lote, o Pyscripter será iniciado, com o caminho correto.

Mais popular que o Pyscripter, o Eclipse é uma escolha comum entre os desenvolvedores. Na seção a seguir, explicaremos como configurá-lo para desenvolver e testar complementos.

16.4.4 Depurando usando Eclipse e PyDev

Instalação

Para usar o Eclipse, verifique se você instalou o seguinte

- Eclipse
- Aptana Studio 3 Plugin ou PyDev
- QGIS 2.x
- Você também pode instalar o **Remote Debug**, um complemento QGIS. No momento, ainda é experimental, então habilite *Complementos experimentais* em *Complementos* → *Gerenciar e Instalar Complementos...* → *Opções* de antemão.

Para preparar seu ambiente para usar o Eclipse no Windows, você também deve criar um arquivo em lotes e usá-lo para iniciar o Eclipse:

1. Localize a pasta em que a `qgis_core.dll` fica. Normalmente, este é `C:\OSGeo4W\apps\qgis\bin`, mas se você compilou seu próprio aplicativo QGIS, vai estar em sua pasta de compilação `output/bin/RelWithDebInfo`
2. Encontre o executável `eclipse.exe`.
3. Crie o script a seguir e use-o para iniciar o eclipse ao desenvolver complementos QGIS.

```
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
```

Configurando o Eclipse

1. No Eclipse, crie um novo projeto. Você pode selecionar *Projeto Geral* e vincular suas fontes reais mais tarde, para que realmente não importe onde você coloca esse projeto.
2. Clique com o botão direito do mouse no seu novo projeto e escolha *Novo → Pasta*.
3. Clique em *Avançado* e escolha *Link para local alternativo (pasta vinculada)*. Caso você já tenha fontes que deseja depurar, escolha estas. Caso contrário, crie uma pasta como já foi explicado.

Agora, na exibição *Project Explorer*, sua árvore de fontes é exibida e você pode começar a trabalhar com o código. Você já possui destaque de sintaxe e todas as outras ferramentas IDE poderosas disponíveis.

Configurando o depurador

Para fazer o depurador funcionar:

1. Alterne para a perspectiva Debug no Eclipse (*Janela → Abrir Perspectiva → Outro → Depurar*).
2. inicie o servidor de depuração do PyDev escolhendo *PyDev → Iniciar Servidor de Depuração*.
3. Agora, o Eclipse está aguardando uma conexão do QGIS com seu servidor de depuração e, quando o QGIS se conectar ao servidor de depuração, permitirá que ele controle os scripts python. Foi exatamente para isso que instalamos o complemento *Remote Debug*. Portanto, inicie o QGIS, caso ainda não o tenha, e clique no símbolo de bug.

Agora você pode definir um ponto de interrupção e, assim que o código o atingir, a execução será interrompida e você poderá inspecionar o estado atual do seu complemento. (O ponto de interrupção é o ponto verde na imagem abaixo, defina um clicando duas vezes no espaço em branco à esquerda da linha em que você deseja definir o ponto de interrupção).

Uma coisa muito interessante que você pode usar agora é o console de depuração. Certifique-se de que a execução esteja atualmente parada em um ponto de interrupção, antes de continuar.

1. Abra a visualização Console (*Janela → Mostrar visualização*). Ele mostrará o console *Servidor de Depuração* que não é muito interessante. Mas existe um botão *Abrir Console* que permite mudar para um console de depuração PyDev mais interessante.
2. Clique na seta ao lado do botão *Abrir Console* e escolha *PyDev Console*. Uma janela se abre para perguntar qual console você deseja iniciar.
3. Escolha *Console de Depuração do PyDev*. Caso esteja acinzentado e diga para você iniciar o depurador e selecionar o quadro válido, verifique se o depurador remoto está conectado e se está atualmente em um ponto de interrupção.

Agora você tem um console interativo que permite testar todos os comandos do contexto atual. Você pode manipular variáveis ou fazer chamadas de API ou o que quiser.

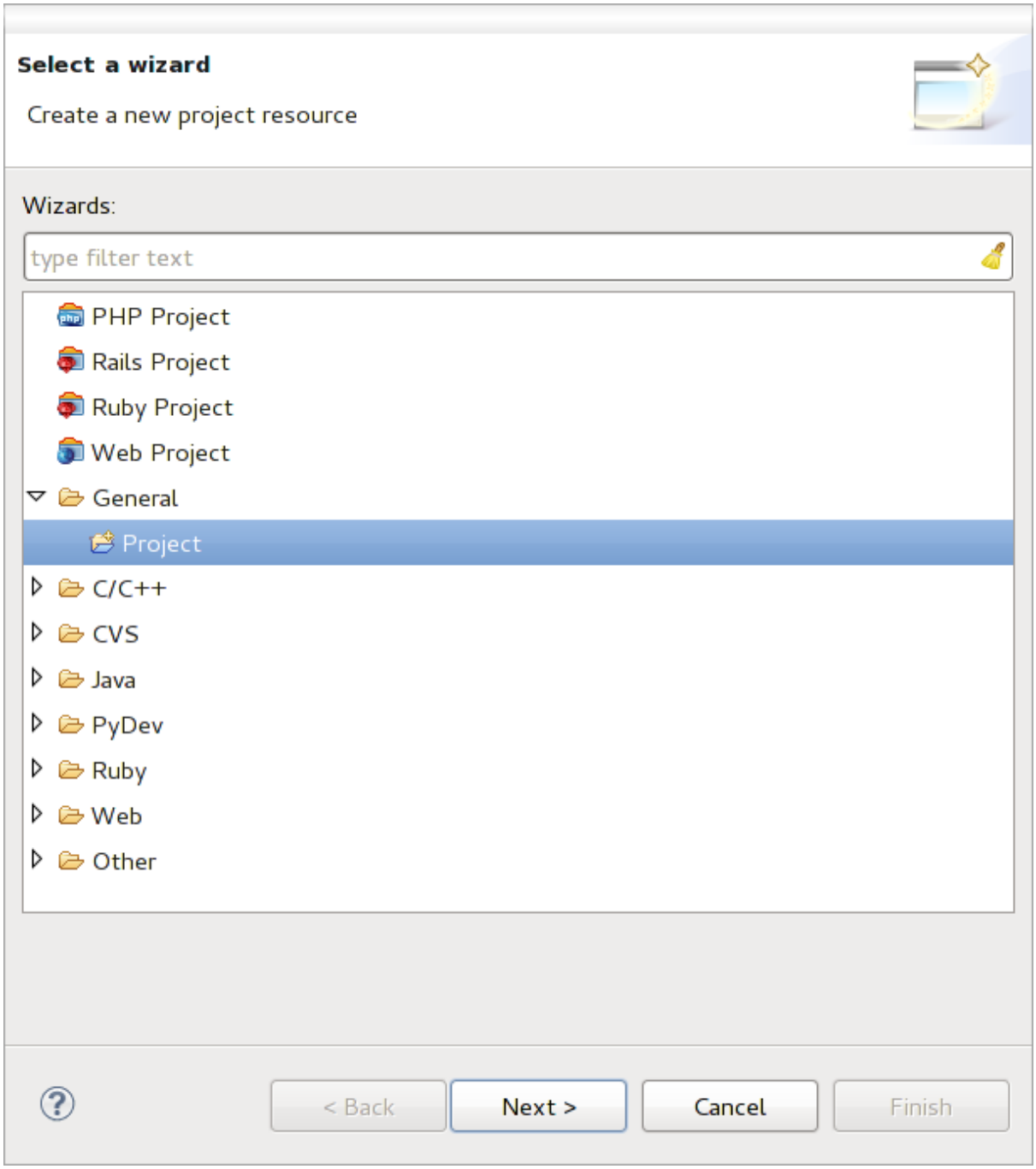


Fig. 16.1: Projeto Eclipse

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

Fig. 16.2: Ponto de interrupção

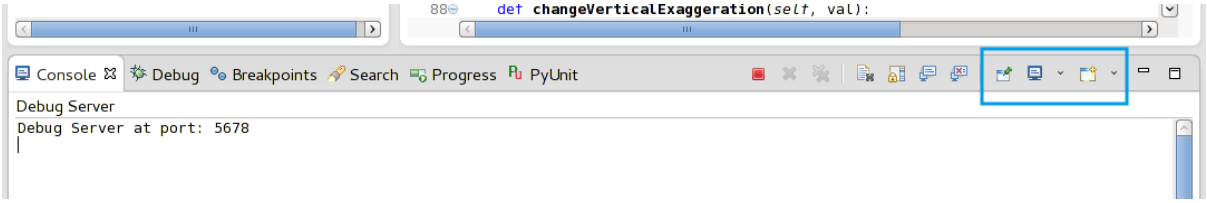


Fig. 16.3: Console de Depuração do PyDev

Dica: Um pouco chato é que, toda vez que você digita um comando, o console volta ao servidor de depuração. Para interromper esse comportamento, você pode clicar no botão *Pin Console* quando estiver na página Debug Server e ela deve se lembrar dessa decisão pelo menos para a sessão de depuração atual.

Fazendo o eclipse entender a API

Um recurso muito útil é que o Eclipse realmente saiba sobre a API QGIS. Isso permite verificar seu código quanto a erros de digitação. Mas não é só isso, também permite que o Eclipse o ajude no preenchimento automático das importações para chamadas de API.

Para fazer isso, o Eclipse analisa os arquivos da biblioteca QGIS e obtém todas as informações disponíveis. A única coisa que você precisa fazer é dizer ao Eclipse onde encontrar as bibliotecas.

1. Clique em *Janela → Preferências → PyDev → Intérprete → Python*.

   Você verá seu interpretador python configurado na parte superior da janela (no momento python2.7 para QGIS) e algumas guias na parte inferior. As guias interessantes para nós são *Libraries* e *Forced Builtins*.

2. Primeiramente abra a guia Libraries.
3. Adicione uma nova pasta e escolha a pasta python da sua instalação do QGIS. Se você não souber onde está essa pasta (não é a pasta plugins):
   1. Abra o QGIS
   2. Inicie um console python
   3. Entre `qgis`
   4. e pressione Enter. Ele mostrará qual módulo QGIS ele usa e seu caminho.
   5. Retire o caminho `/qgis/__init__.pyc` deste caminho e você terá o caminho que está procurando.
4. Você também deve adicionar sua pasta de complementos aqui (está em `python/plugins` na pasta user profile).

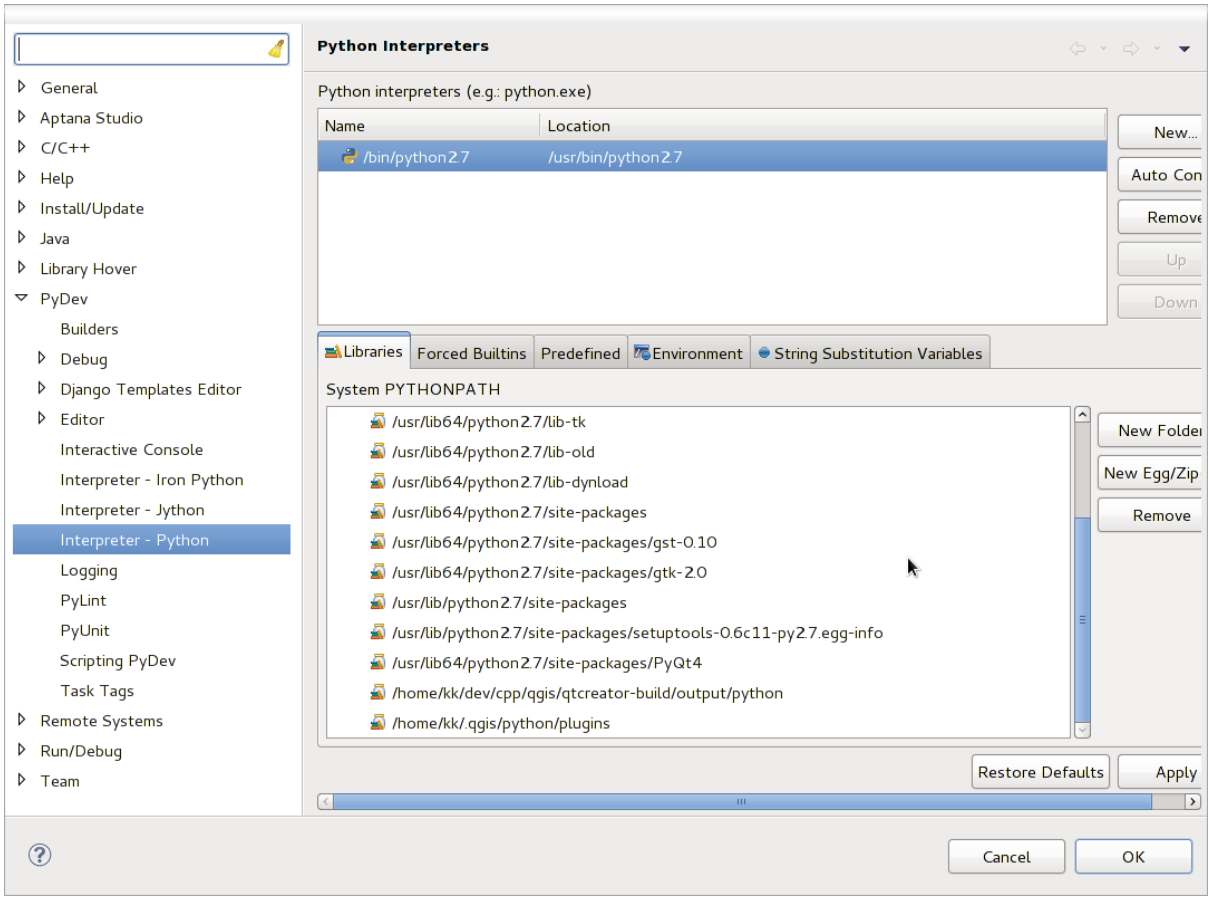


Fig. 16.4: Console de Depuração do PyDev

5. Em seguida, pule para a guia *Forced Builtins*, clique em *Novo...* e digite `qgis`. Isso fará com que o Eclipse analise a API QGIS. Você provavelmente também deseja que o Eclipse conheça a API PyQt. Portanto, adicione também PyQt como forced builtin. Provavelmente, isso já deve estar presente na sua guia de bibliotecas.
6. Clique em *OK* e pronto.

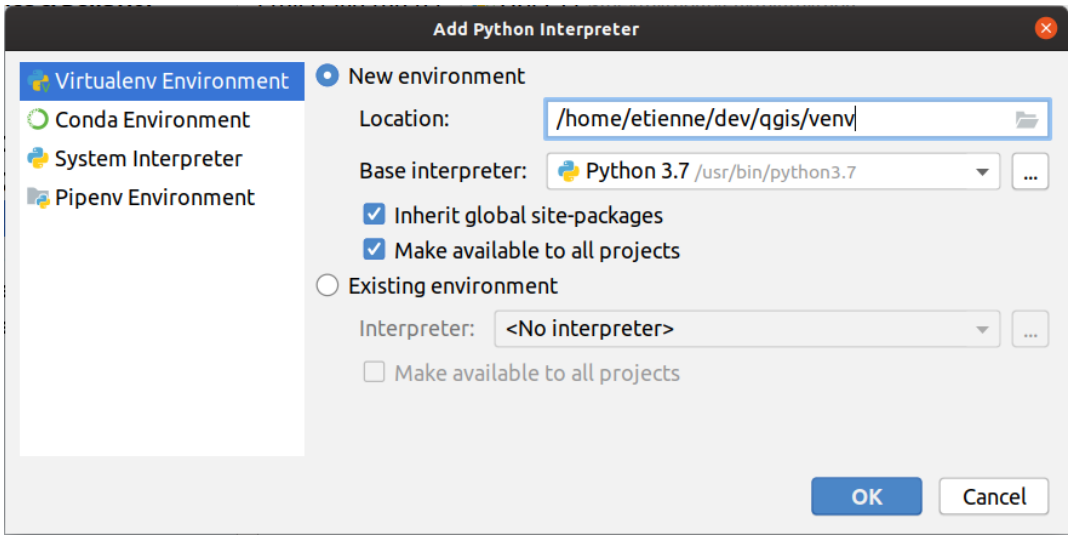
Nota: Toda vez que a API do QGIS for alterada (por exemplo, se você estiver compilando o mestre do QGIS e o arquivo SIP for alterado), volte a esta página e clique em *Aplicar*. Isso permitirá que o Eclipse analise todas as bibliotecas novamente.

16.4.5 Depurando com PyCharm no Ubuntu com um QGIS compilado

PyCharm é um IDE para Python desenvolvido pela JetBrains. Existe uma versão gratuita chamada Community Edition e uma paga, chamada Professional. Você pode fazer o download do PyCharm no website: https://www.jetbrains.com/pycharm/download

Estamos assumindo que você compilou o QGIS no Ubuntu com o diretório de compilação fornecido `~/dev/qgis/build/master`. Não é obrigatório ter um QGIS auto-compilado, mas apenas isso foi testado. Os caminhos devem ser adaptados.

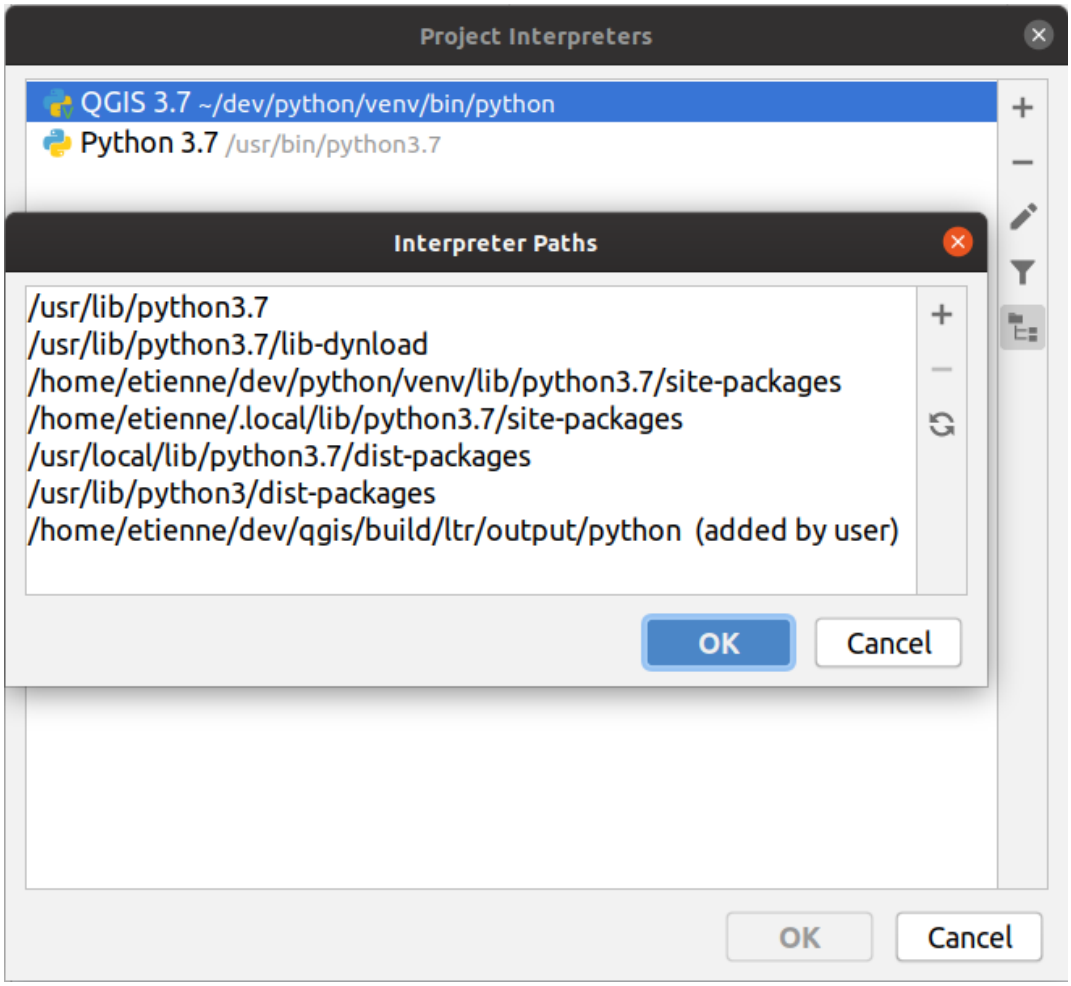
1. No PyCharm, em seu *Propriedades do Projeto*, *Interpretador do Projeto*, criaremos um ambiente virtual Python chamado `QGIS`.
2. Clique na engrenagem pequena e depois *Adicionar*.
3. Selecione *Virtualenv environment*.
4. Select a generic location for all your Python projects such as `~/dev/qgis/venv` because we will use this Python interpreter for all our plugins.
5. Choose a Python 3 base interpreter available on your system and check the next two options *Inherit global site-packages* and *Make available to all projects*.



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

1. Restart PyCharm and you can start using this new Python virtual environment for all your plugins.

PyCharm will be aware of the QGIS API and also of the PyQt API if you use Qt provided by QGIS like `from qgis.PyQt.QtCore import QDir`. The autocompletion should work and PyCharm can inspect your code.


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

In the professional version of PyCharm, remote debugging is working well. For the Community edition, remote debugging is not available. You can only have access to a local debugger, meaning that the code must run *inside* PyCharm (as script or unittest), not in QGIS itself. For Python code running *in* QGIS, you might use the *First Aid* plugin mentioned above.

16.4.6 Debugging using PDB

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

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

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

2. Then run QGIS from the command line.

   On Linux do:

```
$ ./Qgis
```

   On macOS do:

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

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

TODO: Add testing information

16.5 Lançando seu complemento

- *Metadados e nomes*
- *Código e ajuda*
- *Repositório Oficial de complementos Python*
  - *Permissões*
  - *Gerenciamento de confiança*
  - *Validação*
  - *Estrutura do complemento*

Quando o seu complemento estiver pronto e você achar que ele pode ser útil para algumas pessoas, não hesite em enviá-lo para *Repositório Oficial de complementos Python*. Nessa página, você também pode encontrar diretrizes de empacotamento sobre como preparar o complemento para funcionar bem com o instalador do complemento. Ou, caso você queira configurar seu próprio repositório de complementos, crie um arquivo XML simples que listará os complementos e seus metadados.

Por favor, atente especialmente para as seguintes sugestões:

16.5.1 Metadados e nomes

- evite usar um nome muito semelhante aos dos complementos existentes
- se o seu complemento uma funcionalidade semelhante a um complemento existente, explique as diferenças no campo Sobre, para que o usuário saiba qual usar, sem a necessidade de instalá-lo e testá-lo.
- evite repetir "plugin" no nome do próprio complemento
- use o campo de descrição nos metadados para uma descrição de 1 linha, o campo Sobre para obter instruções mais detalhadas
- inclua um repositório de códigos, um rastreador de bug e uma página inicial; isso aumentará muito a possibilidade de colaboração e pode ser feito com muita facilidade com uma das infraestruturas da web disponíveis (GitHub, GitLab, Bitbucket, etc.)
- escolha as tags com cuidado: evite as não informativas (por exemplo, vetor) e prefira as que já são usadas por outras pessoas (consulte o site do complemento)
- adicione um ícone adequado, não deixe o ícone padrão; consulte a interface QGIS para obter uma sugestão do estilo a ser usado

16.5.2 Código e ajuda

- não inclua arquivo gerado (ui_*.py, resources_rc.py, arquivos de ajuda gerados…) e coisas inúteis (por exemplo, .gitignore) no repositório
- adicione o plug-in ao menu apropriado (Vetor, Raster, Web, Banco de dados)
- quando apropriado (complementos que executam análises), considere adicionar o complemento como um subcomplemento da estrutura de Processar: isso permitirá que os usuários o executem em lote, integrem-no em fluxos de trabalho mais complexos e não necessitem projetar uma interface
- inclua pelo menos documentação mínima e, se útil para testar e entender, dados de amostra.

16.5.3 Repositório Oficial de complementos Python

Você pode encontrar o repositório *oficial* de complementos Python em https://plugins.qgis.org/.

Para usar o repositório oficial, você deve obter um ID OSGEO no portal web da OSGEO.

Depois de fazer o upload do seu complemento, ele será aprovado por um membro da equipe e você será notificado.

TODO: Insira um link para o documento de governança

Permissões

Essas regras foram implementadas no repositório oficial de complementos:

- todo usuário registrado pode adicionar um novo complemento
- os usuários da *equipe* podem aprovar ou desaprovar todas as versões do complemento
- usuários com a permissão especial *plugins.can_approve* obtêm as versões que eles enviam automaticamente aprovadas
- usuários com a permissão especial *plugins.can_approve* podem aprovar versões enviadas por outros, desde que estejam na lista dos *proprietários* do complemento
- um complemento específico pode ser excluído e editado apenas por usuários da *equipe* e por seus *proprietários*
- se um usuário sem a permissão *plugins.can_approve* carrega uma nova versão, a versão do complemento é automaticamente não aprovada.

Gerenciamento de confiança

Os membros da equipe podem conceder *confiança* aos criadores de complementos selecionados, configurando a permissão *plugins.can_approve* através do aplicativo front-end.

A visualização de detalhes do complemento oferece links diretos para fornecer confiança ao criador ou ao *proprietário* do complemento.

Validação

Os metadados do complemento são importados e validados automaticamente do pacote compactado quando o upload é feito.

Aqui estão algumas regras de validação que você deve saber quando deseja fazer upload de um complemento no repositório oficial:

1. o nome da pasta principal que contém o complemento deve conter apenas caracteres ASCII (A-Z e a-z), dígitos e os caracteres sublinhados (_) e menos (-), também não pode começar com um dígito
2. `metadata.txt` é necessário
3. todos os metadados necessários, listados em *metadata table* devem estar presentes
4. o campo de metadados da *versão* deve ser exclusivo

Estrutura do complemento

Seguindo as regras de validação, o pacote compactado (.zip) do seu complemento deve ter uma estrutura específica para validar como um complemento funcional. Como o complemento será descompactado na pasta de complementos dos usuários, ele deve ter seu próprio diretório dentro do arquivo .zip para não interferir com outros complementos. Os arquivos obrigatórios são: `metadata.txt` e `__init__.py`. Mas seria bom ter um `README` e, é claro, um ícone para representar o complemento (`resources.qrc`). A seguir, é apresentado um exemplo de como um complemento.zip deve se parecer.

```
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
```

É possível criar complementos na linguagem de programação Python. Em comparação com os complementos clássicos escritos em C++, estes devem ser mais fácil escrever, entender, manter e distribuir devido à natureza dinâmica da linguagem Python.

Os complementos Python são listados junto com os complementos C++ no gerenciador de complementos do QGIS. Eles são pesquisados em `~/(UserProfile)/python/plugins` e nestes caminhos:

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

Para definições de ~ e `(UserProfile)` veja core_and_external_plugins.

Nota: Definindo *QGIS_PLUGINPATH* como um caminho de diretório existente, você pode adicionar esse caminho à lista de caminhos que são procurados por complementos.

CAPÍTULO 17

Escrevendo um complemento de processamento

- *Criando do zero*
- *Atualizando um complemento*

Dependendo do tipo de plug-in que você desenvolverá, pode ser uma opção melhor adicionar sua funcionalidade como um algoritmo de Processar (ou um conjunto deles). Isso proporcionaria uma melhor integração ao QGIS, funcionalidade adicional (já que pode ser executada nos componentes do Processar, como o modelador ou a interface de processamento em lote) e um tempo de desenvolvimento mais rápido (já que o Processar consumirá grande parte do trabalhos).

Para distribuir esses algoritmos, você deve criar um nov complemento que os adicione à Caixa de Ferramentas de Processar. O complemento deve conter um provedor de algoritmos, que deve ser registrado quando o complemento é instanciado.

17.1 Criando do zero

Para criar um complemento do zero que contém um provedor de algoritmos, siga estas etapas usando o Plugin Builder:

1. Instale o complemento **Plugin Builder**
2. Crie um novo complemento usando o Plugin Builder. Quando o Plugin Builder perguntar qual modelo usar, selecione “Processing provider”.
3. O complemento criado contém um provedor com um único algoritmo. O arquivo do provedor e o arquivo do algoritmo são totalmente comentados e contêm informações sobre como modificar o provedor e adicionar algoritmos adicionais. Consulte-os para obter mais informações.

17.2 Atualizando um complemento

Se você deseja adicionar seu complemento existente ao Processar, precisará adicionar algum código.

1. No seu arquivo `metadata.txt`, você precisa adicionar uma variável:

```
hasProcessingProvider=yes
```

2. No arquivo Python em que seu complemento está configurado com o método `initGui`, você precisa adaptar algumas linhas como esta:

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

class YourPluginName():

    def __init__(self):
        self.provider = None

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

    def initGui(self):
        self.initProcessing()

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

3. Você pode criar uma pasta `processing_provider` com três arquivos:
   - `__init__.py` com nada nele. Isso é necessário para criar um pacote Python válido.
   - `provider.py` que criará o Provedor de Processamento e exporá seus algoritmos.

```
from qgis.core import QgsProcessingProvider

from processing_provider.example_processing_algorithm import␣
↪ExampleProcessingAlgorithm


class Provider(QgsProcessingProvider):

    def loadAlgorithms(self, *args, **kwargs):
        self.addAlgorithm(ExampleProcessingAlgorithm())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())

    def id(self, *args, **kwargs):
        """The ID of your plugin, used for identifying the provider.

        This string should be a unique, short, character only string,
        eg "qgis" or "gdal". This string should not be localised.
        """
        return 'yourplugin'

    def name(self, *args, **kwargs):
        """The human friendly name of your plugin in Processing.

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

(continua na próxima página)

(continuação da página anterior)

```
28
29      def icon(self):
30          """Should return a QIcon which is used for your provider inside
31          the Processing toolbox.
32          """
33          return QgsProcessingProvider.icon(self)
```

- `example_processing_algorithm.py` que contém o arquivo de algoritmo de exemplo. Copie/cole o conteúdo do arquivo script template e atualize-o de acordo com suas necessidades.

4. Agora você pode recarregar seu complemento no QGIS e deverá ver o seu exemplo de script na caixa de ferramentas Processar e modelador.

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

CAPÍTULO 18

Biblioteca de análise de rede

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

A biblioteca de análise de rede pode ser usada para:

- criar gráfico matemático a partir de dados geográficos (camadas vetoriais de polilinhas)
- implementar métodos básicos da teoria dos gráficos (atualmente apenas o algoritmo de Dijkstra)

A biblioteca de análise de rede foi criada exportando funções básicas do complemento principal RoadGraph e agora você pode usar seus métodos em complementos ou diretamente do console do Python.

18.1 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)

18.2 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.

Como fonte, podemos usar qualquer camada vetorial de polilinha. Os nós das polilinhas se tornam vértices do gráfico e os segmentos das polilinhas são arestas do gráfico. Se vários nós tiverem as mesmas coordenadas, serão o mesmo vértice do gráfico. Portanto, duas linhas que possuem um nó comum se conectam.

Além disso, durante a criação do gráfico, é possível "fixar" ("vincular") à camada vetorial de entrada qualquer número de pontos adicionais. Para cada ponto adicional, uma correspondência será encontrada - o vértice do gráfico mais próximo ou a aresta do gráfico mais próxima. Neste último caso, a aresta será dividida e um novo vértice será adicionado.

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

Converting from a vector layer to the graph is done using the Builder programming pattern. A graph is constructed using a so-called Director. There is only one Director for now: QgsVectorLayerDirector. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create the graph. Currently, as in the case with the director, only one builder exists: `QgsGraphBuilder`, that creates `QgsGraph` objects. You may want to implement your own builders that will build a graphs compatible with such libraries as BGL or NetworkX.

To calculate edge properties the programming pattern strategy is used. For now only QgsNetworkDistanceStrategy strategy (that takes into account the length of the route) and QgsNetworkSpeedStrategy (that also considers the speed) are availabile. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses a strategy that computes travel time using edge length and speed value from attributes.

It's time to dive into the process.

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

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

Then some examples for creating a director

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

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

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

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

Aqui segue uma lista de todos os significados dos parâmetros:

- `vectorLayer` — vector layer used to build the graph

- `directionFieldId` — index of the attribute table field, where information about roads direction is stored. If `-1`, then don't use this info at all. An integer.
- `directDirectionValue` — field value for roads with direct direction (moving from first line point to last one). A string.
- `reverseDirectionValue` — field value for roads with reverse direction (moving from last line point to first one). A string.
- `bothDirectionValue` — field value for bidirectional roads (for such roads we can move from first point to last and from last to first). A string.
- `defaultDirection` — default road direction. This value will be used for those roads where field `directionFieldId` is not set or has some value different from any of the three values specified above. Possible values are:
  - `QgsVectorLayerDirector.DirectionForward` — One-way direct
  - `QgsVectorLayerDirector.DirectionBackward` — One-way reverse
  - `QgsVectorLayerDirector.DirectionBoth` — Two-way

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

```
# The index of the field that contains information about the edge speed
attributeId = 1
# Default speed value
defaultValue = 50
# Conversion from speed to metric units ('1' means no conversion)
toMetricFactor = 1
strategy = QgsNetworkSpeedStrategy(attributeId, defaultValue, toMetricFactor)
```

And tell the director about this strategy

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', 3)
director.addStrategy(strategy)
```

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

- `crs` — coordinate reference system to use. Mandatory argument.
- `otfEnabled` — use "on the fly" reprojection or no. By default const:*True* (use OTF).
- `topologyTolerance` — topological tolerance. Default value is 0.
- `ellipsoidID` — ellipsoid to use. By default "WGS84".

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

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

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

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

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

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

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

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

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

18.3 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 a 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:

- `source` — input graph
- `startVertexIdx` — index of the point on the tree (the root of the tree)
- `criterionNum` — number of edge property to use (started from 0).

```
tree = QgsGraphAnalyzer.shortestTree(graph, startId, 0)
```

The `dijkstra` method has the same arguments, but returns two arrays. In the first array element *n* contains index of the incoming edge or -1 if there are no incoming edges. In the second array element *n* contains the distance from the root of the tree to vertex *n* or DOUBLE_MAX if vertex *n* is unreachable from the root.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, startId, 0)
```

Here is some very simple code to display the shortest path tree using the graph created with the `shortestTree` method (select linestring layer in *Layers* panel and replace coordinates with your own).

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

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

```

(continua na próxima página)

(continuação da página anterior)

```
pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

i = 0
while (i < tree.edgeCount()):
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
  rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
  i = i + 1
```

Same thing but using the `dijkstra` method

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')

director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
  rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())
```

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

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

Here is the sample code for QGIS Python Console (you may need to load and select a linestring layer in TOC and replace coordinates in the code with yours) that uses the `shortestTree` method

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
builder = QgsGraphBuilder(vectorLayer.sourceCrs())
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)

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

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

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

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

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

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

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

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

(continua na próxima página)

(continuação da página anterior)

```
40
41 # Display
42 rb = QgsRubberBand(iface.mapCanvas())
43 rb.setColor(Qt.green)
44
45 # This may require coordinate transformation if project's CRS
46 # is different than layer's CRS
47 for p in route:
48     rb.addPoint(p)
```

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

```
1  from qgis.core import *
2  from qgis.gui import *
3  from qgis.analysis import *
4
5  from qgis.PyQt.QtCore import *
6  from qgis.PyQt.QtGui import *
7
8  vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
   ↪'lines')
9  director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
   ↪QgsVectorLayerDirector.DirectionBoth)
10 strategy = QgsNetworkDistanceStrategy()
11 director.addStrategy(strategy)
12
13 builder = QgsGraphBuilder(vectorLayer.sourceCrs())
14
15 startPoint = QgsPointXY(1179661.925139,5419188.074362)
16 endPoint = QgsPointXY(1180942.970617,5420040.097560)
17
18 tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
19 tStart, tStop = tiedPoints
20
21 graph = builder.graph()
22 idxStart = graph.findVertex(tStart)
23 idxEnd = graph.findVertex(tStop)
24
25 (tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idxStart, 0)
26
27 if tree[idxEnd] == -1:
28     raise Exception('No route!')
29
30 # Total cost
31 cost = costs[idxEnd]
32
33 # Add last point
34 route = [graph.vertex(idxEnd).point()]
35
36 # Iterate the graph
37 while idxEnd != idxStart:
38     idxEnd = graph.edge(tree[idxEnd]).fromVertex()
39     route.insert(0, graph.vertex(idxEnd).point())
40
41 # Display
42 rb = QgsRubberBand(iface.mapCanvas())
43 rb.setColor(Qt.red)
44
45 # This may require coordinate transformation if project's CRS
46 # is different than layer's CRS
47 for p in route:
```

(continua na próxima página)

(continuação da página anterior)

```
    rb.addPoint(p)
```

18.3.2 Areas of availability

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

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

To find the areas of availability we can use the `dijkstra` method of the `QgsGraphAnalyzer` class. It is enough to compare the elements of the cost array with a predefined value. If cost[i] is less than or equal to a predefined value, then vertex i is inside the area of availability, otherwise it is outside.

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

Aqui está um exemplo

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())


pStart = QgsPointXY(1179661.925139, 5419188.074362)
delta = iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(iface.mapCanvas(), True)
rb.setColor(Qt.green)
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() + delta))

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

idStart = graph.findVertex(tStart)

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

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

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

for i in upperBound:
    centerPoint = graph.vertex(i).point()
```

(continua na próxima página)

(continuação da página anterior)

```
    rb = QgsRubberBand(iface.mapCanvas(), True)
    rb.setColor(Qt.red)
    rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() - delta))
    rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() - delta))
    rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() + delta))
    rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() + delta))
```

CAPÍTULO 19

Servidor QGIS e Python

19.1 Introdução

O Servidor QGIS é três coisas diferentes:

1. Biblioteca do Servidor QGIS: uma biblioteca que fornece uma API para criar serviços da web OGC
2. Servidor QGIS FCGI: um aplicativo binário FCGI `qgis_maserv.fcgi` que, juntamente com um servidor web, implementa um conjunto de serviços OCG (WMS, WFS, WCS etc.) e APIs OGC (WFS3/OAPIF)
3. Servidor de Desenvolvimento QGIS: um aplicativo binário do servidor de desenvolvimento `qgis_mapserver` que implementa um conjunto de serviços OCG (WMS, WFS, WCS etc.) e APIs OGC (WFS3/OAPIF)

Este capítulo do livro de receitas se concentra no primeiro tópico e, ao explicar o uso da API do Servidor QGIS, mostra como é possível usar o Python para estender, aprimorar ou personalizar o comportamento do servidor ou como usar a API do Servidor QGIS para incorporar o servidor QGIS ao outra aplicação.

Existem algumas maneiras diferentes de alterar o comportamento do Servidor QGIS ou estender seus recursos para oferecer novos serviços ou APIs personalizados; estes são os principais cenários que você pode encontrar:

- EMBEDDING → Usa a API do Servidor QGIS de outro aplicativo Python
- STANDALONE → Executa o Servidor QGIS como um serviço WSGI/HTTP independente
- FILTERS → Aprimora/personaliza o Servidor QGIS com complementos de filtro
- SERVICES → Adiciona um novo *SERVICE*
- OGC APIs → Adiciona um novo *OGC API*

Aplicativos de incorporação e independentes exigem o uso da API Python do Servidor QGIS diretamente de outro script ou aplicativo Python, enquanto as opções restantes são mais adequadas para quando você deseja adicionar recursos personalizados a um aplicativo binário padrão do Servidor QGIS (FCGI ou servidor de desenvolvimento): neste caso você precisará escrever um complemento Python para o aplicativo do servidor e registrar seus filtros, serviços ou APIs personalizados.

19.2 Noções básicas da API do servidor

As classes fundamentais envolvidas em um aplicativo típico do Servidor QGIS são:

- `QgsServer` a instância do servidor (geralmente uma instância única para toda a vida do aplicativo)
- :class:`QgsServerRequest <qgis.server.QgsServerRequest>`o objeto de solicitação (normalmente recriado em cada solicitação)
- `QgsServerResponse` o objeto de resposta (normalmente recriado em cada solicitação)
- `QgsServer.handleRequest(request, response)` processa a solicitação e preenche a resposta

O Servidor QGIS FCGI ou o fluxo de trabalho do servidor de desenvolvimento pode ser resumido da seguinte maneira:

```
initialize the QgsApplication
create the QgsServer
the main server loop waits forever for client requests:
    for each incoming request:
        create a QgsServerRequest request
        create a QgsServerResponse response
        call QgsServer.handleRequest(request, response)
            filter plugins may be executed
        send the output to the client
```

Dentro do método `QgsServer.handleRequest (request, response)`, os callbacks dos complementos de filtro são chamados e `QgsServerRequest` e `QgsServerResponse` são disponibilizados para os complementos através da `QgsServerInterface`.

Aviso: As classes de servidor QGIS não são seguras para threads, você sempre deve usar um modelo ou contêineres de multiprocessamento ao criar aplicativos escaláveis com base na API do Servidor QGIS.

19.3 Independente ou incorporado

Para servidor autônomo ou aplicativos integrados, você precisará usar as classes de servidor mencionadas acima diretamente, agrupando-as em uma implementação de servidor da Web que gerencia todas as interações do protocolo HTTP com o cliente.

Um exemplo mínimo do uso da API do Servidor QGIS (sem a parte HTTP) é a seguir:

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

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

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

# Create a response objects
response = QgsBufferServerResponse()

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

(continua na próxima página)

(continuação da página anterior)

```
20
21 print(response.headers())
22 print(response.body().data().decode('utf8'))
23
24 app.exitQgis()
```

Aqui está um exemplo de aplicativo autônomo completo desenvolvido para o teste de integração contínua no repositório de código-fonte QGIS, ele mostra um amplo conjunto de filtros de complementos diferentes e esquemas de autenticação (não significa para produção porque eles foram desenvolvidos apenas para fins de teste, mas ainda interessantes para aprendizagem) :

https://github.com/qgis/QGIS/blob/master/tests/src/python/qgis_wrapped_server.py

19.4 Complementos do servidor

Os complementos de servidor python são carregados uma vez quando o aplicativo Servidor QGIS é iniciado e podem ser usados para registrar filtros, serviços ou APIs.

A estrutura de um complemento de servidor é muito semelhante à sua contraparte na área de trabalho, um objeto `QgsServerInterface` é disponibilizado para os complemento e os complemento podem registrar um ou mais filtros, serviços ou APIs personalizados para o registro correspondente usando um dos métodos expostos pela interface do servidor.

19.4.1 Complementos de filtro de servidor

Filters come in three different flavors and they can be instanciated by subclassing one of the classes below and by calling the corresponding method of `QgsServerInterface`:

Tipo de Filtro	Classe Básica	Registro da QgsServerInterface
I/O	`QgsServerFilter`	`registerFilter`
Controle de Acesso	`QgsAccessControlFilter`	`registerAccessControl`
Cache	`QgsServerCacheFilter`	`registerServerCache`

Filtros I/O

Os filtros I/O podem modificar a entrada e saída do servidor (a solicitação e a resposta) dos serviços principais (WMS, WFS etc.), permitindo fazer qualquer tipo de manipulação do fluxo de trabalho dos serviços; é possível, por exemplo, restringir o acesso nas camadas selecionadas, injetar uma folha de estilo XSL na resposta XML, adicionar uma marca d'água a uma imagem WMS gerada e assim por diante.

A partir deste ponto, você pode achar útil uma rápida olhada no server plugins API docs.

Cada filtro deve implementar pelo menos um dos três retornos de chamada:

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

Todos os filtros têm acesso ao objeto de solicitação/resposta (`QgsRequestHandler`) e podem manipular todas as suas propriedades (entrada/saída) e gerar exceções (embora de uma maneira bastante específica, como iremos ver abaixo).

Here is the pseudo code showing how the server handles a typical request and when the filter's callbacks are called:

```
for each incoming request:
    create GET/POST request handler
    pass request to an instance of QgsServerInterface
    call requestReady filters
    if there is not a response:
        if SERVICE is WMS/WFS/WCS:
            create WMS/WFS/WCS service
            call service's executeRequest
                possibly call sendResponse for each chunk of bytes
                sent to the client by a streaming services (WFS)
        call responseComplete
        call sendResponse
    request handler sends the response to the client
```

The following paragraphs describe the available callbacks in details.

requestReady

This is called when the request is ready: incoming URL and data have been parsed and before entering the core services (WMS, WFS etc.) switch, this is the point where you can manipulate the input and perform actions like:

- autenticação/autorização
- redirects
- add/remove certain parameters (typenames for example)
- raise exceptions

You could even substitute a core service completely by changing **SERVICE** parameter and hence bypassing the core service completely (not that this make much sense though).

sendResponse

This is called whenever any output is sent to **FCGI** `stdout` (and from there, to the client), this is normally done after core services have finished their process and after responseComplete hook was called, but in a few cases XML can become so huge that a streaming XML implementation was needed (WFS GetFeature is one of them), in this case, `sendResponse` is called multiple times before the response is complete (and before `responseComplete` is called). The obvious consequence is that `sendResponse` is normally called once but might be exceptionally called multiple times and in that case (and only in that case) it is also called before `responseComplete`.

`sendResponse` é o melhor local para manipulação direta da saída do serviço principal e enquanto `responseComplete` tipicamente também é uma opção, `sendResponse` é a única opção viável no caso de serviços de streaming.

responseComplete

Isso é chamado uma vez quando os serviços principais (se atingidos) concluem o processo e a solicitação está pronta para ser enviada ao cliente. Como discutido acima, isso normalmente é chamado antes `sendResponse` exceto para serviços de streaming (ou outros filtros de complementos) que podem ter chamado :meth:`sendResponse <qgis.server.QgsServerFilter.sendResponse> `anteriormente.

`responseComplete` é o local ideal para fornecer a implementação de novos serviços (WPS ou serviços personalizados) e executar a manipulação direta da saída proveniente dos serviços principais (por exemplo, para adicionar uma marca d'água em uma imagem WMS).

Raising exceptions from a plugin

Some work has still to be done on this topic: the current implementation can distinguish between handled and unhandled exceptions by setting a `QgsRequestHandler` property to an instance of QgsMapServiceException, this way the main C++ code can catch handled python exceptions and ignore unhandled exceptions (or better: log them).

This approach basically works but it is not very "pythonic": a better approach would be to raise exceptions from python code and see them bubbling up into C++ loop for being handled there.

Escrevendo um complemento de servidor

A server plugin is a standard QGIS Python plugin as described in *Desenvolvendo complementos Python*, that just provides an additional (or alternative) interface: a typical QGIS desktop plugin has access to QGIS application through the `QgisInterface` instance, a server plugin has only access to a `QgsServerInterface` when it is executed within the QGIS Server application context.

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

```
server=True
```

Importante: Only plugins that have the `server=True` metadata set will be loaded and executed by QGIS Server.

The example plugin discussed here (with many more) is available on github at https://github.com/elpaso/qgis3-server-vagrant/tree/master/resources/web/plugins, a few server plugins are also published in the official QGIS plugins repository.

Arquivos de complementos

Aqui está a estrutura de diretórios do nosso complemento de servidor de exemplo

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

__init__.py

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

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

HelloServer.py

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

A server plugin typically consists in one or more callbacks packed into instances of a `QgsServerFilter`.

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

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

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

```
class HelloFilter(QgsServerFilter):

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

    def requestReady(self):
        QgsMessageLog.logMessage("HelloFilter.requestReady")

    def sendResponse(self):
        QgsMessageLog.logMessage("HelloFilter.sendResponse")

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

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

```
class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(), 100)
```

The second parameter of `registerFilter` sets a priority which defines the order for the callbacks with the same name (the lower priority is invoked first).

By using the three callbacks, plugins can manipulate the input and/or the output of the server in many different ways. In every moment, the plugin instance has access to the `QgsRequestHandler` through the `QgsServerInterface`. The `QgsRequestHandler` class has plenty of methods that can be used to alter the input parameters before entering the core processing of the server (by using `requestReady()`) or after the request has been processed by the core services (by using `sendResponse()`).

The following examples cover some common use cases:

Modifying the input

The example plugin contains a test example that changes input parameters coming from the query string, in this example a new parameter is injected into the (already parsed) `parameterMap`, this parameter is then visible by core services (WMS etc.), at the end of core services processing we check that the parameter is still there:

```
class ParamsFilter(QgsServerFilter):

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

    def requestReady(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.responseComplete")
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete")
```

This is an extract of what you see in the log file:

```
 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪HelloServerServer - loading filter ParamsFilter
 src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0]␣
↪Server plugin HelloServer loaded!
 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0]␣
↪Server python plugins loaded
 src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms]␣
↪inserting pair SERVICE // HELLO into the parameter map
 src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter␣
↪plugin default requestReady called
 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]␣
↪SUCCESS - ParamsFilter.responseComplete
```

On the highlighted line the “SUCCESS” string indicates that the plugin passed the test.

The same technique can be exploited to use a custom service instead of a core one: you could for example skip a **WFS SERVICE** request or any other core request just by changing the **SERVICE** parameter to something different and the core service will be skipped, then you can inject your custom results into the output and send them to the client (this is explained here below).

Dica: If you really want to implement a custom service it is recommended to subclass `QgsService` and register your service on `registerFilter` by calling its `registerService(service)`

Modifying or replacing the output

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

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

class WatermarkFilter(QgsServerFilter):

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

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (params.get('SERVICE').upper() == 'WMS' \
                and params.get('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image ready %s" % request.parameter("FORMAT"))
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG" if "png" in request.parameter("FORMAT") else "JPG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
```

In this example the **SERVICE** parameter value is checked and if the incoming request is a **WMS GETMAP** and no exceptions have been set by a previously executed plugin or by the core service (WMS in this case), the WMS generated image is retrieved from the output buffer and the watermark image is added. The final step is to clear the output buffer and replace it with the newly generated image. Please note that in a real-world situation we should also check for the requested image type instead of supporting PNG or JPG only.

Access control filters

Access control filters gives the developer a fine-grained control over which layers, features and attributes can be accessed, the following callbacks can be implemented in an access control filter:

- `layerFilterExpression(layer)`
- `layerFilterSubsetString(layer)`
- `layerPermissions(layer)`
- `authorizedLayerAttributes(layer, attributes)`
- `allowToEdit(layer, feature)`
- `cacheKey()`

Arquivos de complementos

Here's the directory structure of our example plugin:

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

__init__.py

Este arquivo é requerido pelo sistema de importação do Python. Como para todos os complementos de servidor QGIS, este arquivo contém uma função `serverClassFactory()`, chamada quando o complemento é carregado no QGIS Server na inicialização. Ele recebe uma referência a uma instância de `QgsServerInterface` e deve retornar uma instância da classe do seu complemento. É assim que o exemplo do complemento `__init__.py` se parece com:

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

AccessControl.py

```
class AccessControlFilter(QgsAccessControlFilter):

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

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

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

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

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

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

    def cacheKey(self):
        return super().cacheKey()

class AccessControlServer:

   def __init__(self, serverIface):
      """ Register AccessControlFilter """
      serverIface.registerAccessControl(AccessControlFilter(self.serverIface), 100)
```

This example gives a full access for everybody.

É papel do complemento saber quem está conectado.

On all those methods we have the layer on argument to be able to customise the restriction per layer.

layerFilterExpression

Usado para adicionar uma Expressão para limitar os resultados, por exemplo:

```
def layerFilterExpression(self, layer):
    return "$role = 'user'"
```

To limit on feature where the attribute role is equals to “user”.

layerFilterSubsetString

Same than the previous but use the `SubsetString` (executed in the database)

```
def layerFilterSubsetString(self, layer):
    return "role = 'user'"
```

To limit on feature where the attribute role is equals to “user”.

layerPermissions

Limit the access to the layer.

Return an object of type `LayerPermissions`, which has the properties:

- `canRead` to see it in the `GetCapabilities` and have read access.
- `canInsert` to be able to insert a new feature.
- `canUpdate` to be able to update a feature.
- `canDelete` to be able to delete a feature.

Exemplo:

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

To limit everything on read only access.

authorizedLayerAttributes

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

The argument attribute return the current set of visible attributes.

Exemplo:

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

To hide the ‘role’ attribute.

allowToEdit

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

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

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.

cacheKey

QGIS server maintain a cache of the capabilities then to have a cache per role you can return the role in this method. Or return `None` to completely disable the cache.

19.4.2 Custom services

In QGIS Server, core services such as WMS, WFS and WCS are implemented as subclasses of `QgsService`.

To implemented a new service that will be executed when the query string parameter `SERVICE` matches the service name, you can implemented your own `QgsService` and register your service on the `serviceRegistry` by calling its `registerService(service)`.

Here is an example of a custom service named CUSTOM:

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

class CustomServiceService(QgsService):

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

    def name(self):
        return "CUSTOM"

    def version(self):
        return "1.0.0"

    def allowMethod(method):
        return True

    def executeRequest(self, request, response, project):
        response.setStatusCode(200)
        QgsMessageLog.logMessage('Custom service executeRequest')
        response.write("Custom service executeRequest")


class CustomService():

    def __init__(self, serverIface):
        serverIface.serviceRegistry().registerService(CustomServiceService())
```

19.4.3 Custom APIs

In QGIS Server, core OGC APIs such OAPIF (aka WFS3) are implemented as collections of `QgsServerOgcApiHandler` subclasses that are registered to an instance of `QgsServerOgcApi` (or it's parent class `QgsServerApi`).

To implemented a new API that will be executed when the url path matches a certain URL, you can implemented your own `QgsServerOgcApiHandler` instances, add them to an `QgsServerOgcApi` and register the API on the `serviceRegistry` by calling its `registerApi(api)`.

Here is an example of a custom API that will be executed when the URL contains `/customapi`:

```
import json
import os

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

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


class CustomApiHandler(QgsServerOgcApiHandler):

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

    def path(self):
        return QRegularExpression("/customapi")

    def operationId(self):
        return "CustomApiXYCircle"

    def summary(self):
        return "Creates a circle around a point"

    def description(self):
        return "Creates a circle around a point"

    def linkTitle(self):
        return "Custom Api XY Circle"

    def linkType(self):
        return QgsServerOgcApi.data

    def handleRequest(self, context):
        """Simple Circle"""

        values = self.values(context)
```

(continua na próxima página)

(continuação da página anterior)

```
        x = values['x']
        y = values['y']
        r = values['r']
        f = QgsFeature()
        f.setAttributes([x, y, r])
        f.setGeometry(QgsCircle(QgsPoint(x, y), r).toCircularString())
        exporter = QgsJsonExporter()
        self.write(json.loads(exporter.exportFeature(f)), context)

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

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


class CustomApi():

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

Os trechos de código desta página precisam das seguintes importações se você estiver fora do console do pyqgis:

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

from qgis.core import (
    QgsProject,
    QgsLayerTreeModel,
)

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

CAPÍTULO 20

Folha de dicas para PyQGIS

20.1 Interface de usuário

Alterar aparência

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
app.setStyleSheet(".QWidget {color: blue; background-color: yellow;}")
# You can even read the stylesheet from a file
with  open("testdata/file.qss") as qss_file_content:
    app.setStyleSheet(qss_file_content.read())
```

Alterar ícone e título

```
from qgis.PyQt.QtGui import QIcon

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

20.2 Configurações

Get QgsSettings list

```
from qgis.core import QgsSettings

qs = QgsSettings()

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

20.3 Barra de ferramentas

Remover barra de ferramenas

```
toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)

# and add again
parent.addToolBar(toolbar)
```

Remover barra de ferramentas de ações

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

20.4 Menus

Remover menu

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

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

20.5 Tela

Tela de acesso

```
canvas = iface.mapCanvas()
```

Alterar cor da tela

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

Intervalo de Atualização do Mapa

```
from qgis.core import QgsSettings
# Set milliseconds (150 milliseconds)
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

20.6 Camadas

Adicionar camada vetorial

```
layer = iface.addVectorLayer("testdata/airports.shp", "layer name you like", "ogr")
if not layer or not layer.isValid():
    print("Layer failed to load!")
```

Get active layer

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

Lista todas as camadas

```
from qgis.core import QgsProject

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

Obtém os nomes das camadas

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

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

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

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

Otherwise

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

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

Encontrar camada por nome

```
from qgis.core import QgsProject

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

```
layer name you like
```

Definir camada ativa

```
from qgis.core import QgsProject

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

Refresh layer at interval

```
from qgis.core import QgsProject

```

(continua na próxima página)

(continuação da página anterior)

```
layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)
# Enable auto refresh
layer.setAutoRefreshEnabled(True)
```

Mostrar métodos

```
dir(layer)
```

Adding new feature with feature form

```
from qgis.core import QgsFeature, QgsGeometry

feat = QgsFeature()
geom = QgsGeometry()
feat.setGeometry(geom)
feat.setFields(layer.fields())

iface.openFeatureForm(layer, feat, False)
```

Adding new feature without feature form

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

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

Get features

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

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

Obter feições selecionadas

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

Get selected features Ids

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

Create a memory layer from selected features Ids

```
from qgis.core import QgsFeatureRequest

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

Get geometry

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

```
10.000000, 10.000000
```

Move geometry

```
from qgis.core import QgsFeature, QgsGeometry
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POINT(7 45)")
geom.translate(1, 1)
poly.setGeometry(geom)
print(poly.geometry())
```

```
<QgsGeometry: Point (8 46)>
```

Definir o SRC

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

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

Ver o SRC

```
from qgis.core import QgsProject

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

Ocultar uma coluna de campo

```
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
```

Camada de WKT

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

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

Load all vector layers from GeoPackage

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

(continua na próxima página)

(continuação da página anterior)

```
for subLayer in subLayers:
    name = subLayer.split('!!::!!')[1]
    uri = "%s|layername=%s" % (fileName, name,)
    # Create layer
    sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
    # Add layer to map
    QgsProject.instance().addMapLayer(sub_vlayer)
```

Load tile layer (XYZ-Layer)

```
from qgis.core import QgsRasterLayer, QgsProject

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

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

Remover todas as camadas

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

Remover tudo

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

20.7 Table of contents

Access checked layers

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

Remove contextual menu

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

20.8 Advanced TOC

Root node

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

root = QgsProject.instance().layerTreeRoot()
node_group = root.addGroup("My Group")

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

```

(continua na próxima página)

(continuação da página anterior)

```
node_group.addLayer(layer)

print(root)
print(root.children())
```

Access the first child node

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

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

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

Encontrar grupos e nós

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer

def get_group_layers(group):
   print('- group: ' + group.name())
   for child in group.children():
      if isinstance(child, QgsLayerTreeGroup):
         # Recursive call to get nested groups
         get_group_layers(child)
      else:
         print('  - layer: ' + child.name())


root = QgsProject.instance().layerTreeRoot()
for child in root.children():
   if isinstance(child, QgsLayerTreeGroup):
      get_group_layers(child)
   elif isinstance(child, QgsLayerTreeLayer):
      print ('- layer: ' + child.name())
```

```
- group: My Group
  - layer: layer name you like
```

Encontrar grupo pelo nome

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

```
<qgis._core.QgsLayerTreeGroup object at 0x7fd75560cee8>
```

Encontrar camada por id

```
print(root.findLayer(layer.id()))
```

```
<qgis._core.QgsLayerTreeLayer object at 0x7f56087af288>
```

Adicionar camada

```
from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
# Remove it
QgsProject.instance().removeMapLayer(layer1)
```

Adicionar grupo

```
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
QgsProject.instance().mapLayersByName("layer name you like")[0]
```

Mover camada carregada

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

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

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

parent.removeChildNode(myLayer)
```

Mover camada carregada para um grupo específico

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

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

Alterando a visibilidade

```
myGroup.setItemVisibilityChecked(False)
myLayer.setItemVisibilityChecked(False)
```

Is group selected

```
def isMyGroupSelected( groupName ):
    myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
    return myGroup in iface.layerTreeView().selectedNodes()

print(isMyGroupSelected( 'my group name' ))
```

```
False
```

Expand node

```
print(myGroup.isExpanded())
myGroup.setExpanded(False)
```

Hidden node trick

```
from qgis.core import QgsProject

model = iface.layerTreeView().layerTreeModel()
ltv = iface.layerTreeView()
root = QgsProject.instance().layerTreeRoot()

layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
node = root.findLayer( layer.id())

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

Node signals

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

def onAddedChildren(node, indexFrom, indexTo):
    print ("ADDED", node, indexFrom, indexTo)

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

Remover camada

```
root.removeLayer(layer)
```

Remover grupo

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

Criar novo sumário (TOC)

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

Mover nó

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

Rename node

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

20.9 Processing algorithms

Get algorithms list

```
from qgis.core import QgsApplication

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

```
QGIS (native c++):buffer --> Buffer
```

Get algorithms help

Seleção aleatória

```
from qgis import processing
processing.algorithmHelp("native:buffer")
```

```
...
```

Executar o algoritmo

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

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

```
Processing(0): Results: {'OUTPUT': 'output_d27a2008_970c_4687_b025_f057abbd7319'}
```

How many algorithms are there?

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

How many providers are there?

```
from qgis.core import QgsApplication

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

How many expressions are there?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

20.10 Decorators

CopyRight

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

mQFont = "Sans Serif"
mQFontsize = 9
mLabelQString = "© QGIS 2019"
```

(continua na próxima página)

(continuação da página anterior)

```
mMarginHorizontal = 0
mMarginVertical = 0
mLabelQColor = "#FF0000"

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

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

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

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

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

    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:
```

(continua na próxima página)

(continuação da página anterior)

```
        # Bottom Center
        yOffset = deviceHeight - yOffset - size.height()
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

# Emitted when the canvas has rendered
iface.mapCanvas().renderComplete.connect(_on_render_complete)
# Repaint the canvas map
iface.mapCanvas().refresh()
```

20.11 Composer

Get print layout by name

```
composerTitle = 'MyComposer' # Name of the composer

project = QgsProject.instance()
projectLayoutManager = project.layoutManager()
layout = projectLayoutManager.layoutByName(composerTitle)
```

20.12 Sources

- QGIS Python (PyQGIS) API
- QGIS C++ API
- StackOverFlow QGIS questions
- Script de Klas Karlsson