



PyQGIS 3.40 developer cookbook

QGIS Project

18 abr. 2025

1	Introdução	3
1.1	Escrevendo Scripts no Terminal Python	4
1.2	Complementos Python	4
1.2.1	Plugins de processamento	5
1.3	Executando o código Python quando o QGIS é iniciado	5
1.3.1	O arquivo <code>startup.py</code>	5
1.3.2	A variável de ambiente <code>PYQGIS_STARTUP</code>	5
1.3.3	The <code>--code</code> parameter	5
1.3.4	Additional arguments for Python	6
1.4	Aplicações Python	6
1.4.1	Usando PyQGIS em scripts autônomos	6
1.4.2	Usando o PyQGIS em aplicativos personalizados	7
1.4.3	Executando aplicativos personalizados	8
1.5	Technical notes on PyQt and SIP	8
2	Carregando projetos	9
2.1	Resolvendo caminhos errados	10
2.2	Uso de referências para acelerar as coisas	11
3	Carregando Camadas	13
3.1	Camadas Vetoriais	13
3.2	Camadas Matriciais	16
3.3	Instância <code>QgsProject</code>	18
4	Acessando o sumário (TOC)	19
4.1	A classe <code>QgsProject</code>	19
4.2	Classe <code>QgsLayerTreeGroup</code>	20
5	Usando Camadas Raster	23
5.1	Detalhes da Camada	23
5.2	Renderizador	24
5.2.1	Rasters com Banda Única	25
5.2.2	Rasters Multibandas	25
5.3	Valores de Consulta	26
5.4	Edição de dados raster	26
6	Usando Camadas Vetoriais	27
6.1	Recuperando informações sobre atributos	28
6.2	Iterando sobre Camada Vetorial	28
6.3	Selecionando características	29
6.3.1	Acessando atributos	30

6.3.2	Iteração sobre os feições selecionadas	30
6.3.3	Iterando sobre um subconjunto de feições	31
6.4	Modificando Camadas Vetoriais	32
6.4.1	Adicionar feições	32
6.4.2	Excluir feições	33
6.4.3	Modificar Feições	33
6.4.4	Modificando Camadas Vetoriais com um Buffer	33
6.4.5	Adicionando e Removendo Campos	35
6.5	Utilizando Índices Espaciais	35
6.6	The QgsVectorLayerUtils class	36
6.7	Criando Camadas Vetoriais	37
6.7.1	De uma instância de QgsVectorFileWriter	37
6.7.2	Diretamente das feições	38
6.7.3	De uma instância de QgsVectorLayer	39
6.8	Aparencia (Simbologia) de Camadas de Vetor	40
6.8.1	Renderizador de símbolo único	41
6.8.2	Renderizador de Símbolo Categorizado	42
6.8.3	Renderizador de Símbolo Graduado	43
6.8.4	Trabalhando com Símbolos	44
6.8.5	Criando Renderizadores Personalizados	47
6.9	Outros Tópicos	49
7	Manipulação Geométrica	51
7.1	Construção de Geometria	52
7.2	Acesso a Geometria	52
7.3	Operações e Predicados Geométricos	54
8	Suporte a projeções	57
8.1	Sistemas de Referencia de Coordenadas	57
8.2	Transformação de SRC	59
9	Usando a Tela do Mapa	61
9.1	Incorporar o Mapa da Tela	62
9.2	Bandas raster e fazedor de vértices	63
9.3	Usando Ferramentas de Mapas na Tela	64
9.3.1	Select a feature using QgsMapToolIdentifyFeature	65
9.3.2	Adicionar itens ao menu contextual da tela do mapa	65
9.4	Desenhar ferramenta de mapa personalizada	65
9.5	Desenhar itens da tela do mapa	67
10	Renderização em impressão de mapas	69
10.1	Renderização simples	70
10.2	Renderizando camadas com CRS diferente	70
10.3	Saída usando layout de impressão	71
10.3.1	Checking layout validity	72
10.3.2	Exportando o layout	73
10.3.3	Exportando um atlas como layout	74
11	Expressões, filtragem e cálculo dos valores	75
11.1	Analisando Expressões	76
11.2	Avaliando expressões	76
11.2.1	Expressões Básicas	77
11.2.2	Expressões com feições	77
11.2.3	Filtrando uma camada com expressões	78
11.3	Manipulando erros de expressão	79
12	Leitura e Armazenamento de Configurações	81
13	Comunicação com o usuário	85

13.1	Mostrando mensagens. A classe <code>QgsMessageBar</code>	85
13.2	Mostrando progresso	88
13.3	Carregando	89
13.3.1	<code>QgsMessageLog</code>	89
13.3.2	O python construído no módulo de extração	90
14	Infraestrutura de autenticação	91
14.1	Introdução	92
14.2	Glossário	92
14.3	<code>QgsAuthManager</code> o ponto de entrada	93
14.3.1	Inicie o gerente e defina a master password	93
14.3.2	Preencha o authdb com uma nova entrada de Configuração de Autenticação	93
14.3.3	Remover uma entrada de authdb	95
14.3.4	Leave authcfg expansion to <code>QgsAuthManager</code>	95
14.4	Adapte complementos para usar a infraestrutura de autenticação	96
14.5	GUIs de autenticação	96
14.5.1	GUI para selecionar credenciais	96
14.5.2	GUI do Editor de Autenticação	97
14.5.3	GUI do Editor de Autoridades	98
15	Tarefas - trabalho pesado em segundo plano	99
15.1	Introdução	99
15.2	Exemplos	101
15.2.1	Estendendo <code>QgsTask</code>	101
15.2.2	Tarefa da função	103
15.2.3	Tarefa de um algoritmo de processamento	104
16	Desenvolvendo complementos Python	107
16.1	Estruturando Complementos Python	107
16.1.1	Getting started	107
16.1.2	Writing plugin code	108
16.1.3	Documenting plugins	113
16.1.4	Translating plugins	113
16.1.5	Sharing your plugin	115
16.1.6	Dicas e truques	115
16.2	Partes de código	116
16.2.1	Como chamar um método por um atalho de teclas	117
16.2.2	How to reuse QGIS icons	117
16.2.3	Interface para plugin na caixa de diálogo de opções	117
16.2.4	Embed custom widgets for layers in the layer tree	119
16.3	Configurações de IDE para gravar e depurar complementos	120
16.3.1	Complementos úteis para escrever complementos Python	120
16.3.2	Uma observação sobre como configurar seu IDE no Linux e Windows	120
16.3.3	Depurando usando o Pyscripter IDE (Windows)	120
16.3.4	Depurando usando Eclipse e PyDev	121
16.3.5	Depurando com PyCharm no Ubuntu com um QGIS compilado	125
16.3.6	Debugging using PDB	126
16.4	Lançando seu complemento	127
16.4.1	Metadados e nomes	127
16.4.2	Código e ajuda	127
16.4.3	Repositório Oficial de complementos Python	128
17	Escrevendo um complemento de processamento	131
17.1	Criando do zero	131
17.2	Atualizando um complemento	131
18	Usando Camadas de Complementos	135
18.1	Subclassificação <code>QgsPluginLayer</code>	135

19	Biblioteca de análise de rede	137
19.1	Informação Geral	137
19.2	Elaborando um gráfico	138
19.3	Análise de Gráficos	140
19.3.1	Encontrando os caminhos mais curtos	142
19.3.2	Áreas de disponibilidade	144
20	Servidor QGIS e Python	147
20.1	Introdução	147
20.2	Noções básicas da API do servidor	148
20.3	Independente ou incorporado	148
20.4	Complementos do servidor	149
20.4.1	Complementos de filtro de servidor	149
20.4.2	Serviços personalizados	157
20.4.3	Custom APIs	158
21	Folha de dicas para PyQGIS	161
21.1	Interface de usuário	161
21.2	Configurações	162
21.3	Barra de ferramentas	162
21.4	Menus	162
21.5	Tela	163
21.6	Camadas	163
21.7	Tabela de conteúdos	167
21.8	Advanced TOC	167
21.9	Algoritmos de processamento	170
21.10	Decoradores	171
21.11	Compositor	172
21.12	Sources	172

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.

É concedida a permissão para copiar, distribuir e / ou modificar este documento sob os termos da Licença de Documentação Livre GNU, Versão 1.3 ou qualquer versão posterior publicada pela Fundação de Software Livre; sem Secções Invariantes, sem textos de Capa e sem contracapa.

A cópia da licença é incluída na secção `gnu_fdl`.

Esta licença também se aplica a todos os trechos de código deste documento.

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 secções seguintes):

- Emita comandos no console Python no QGIS
- Crie e use complementos
- Executar automaticamente o código Python quando o QGIS for iniciado
- Criar algoritmos de processamento
- Criar funções para expressões no QGIS
- Crie aplicativos personalizados com base na API QGIS

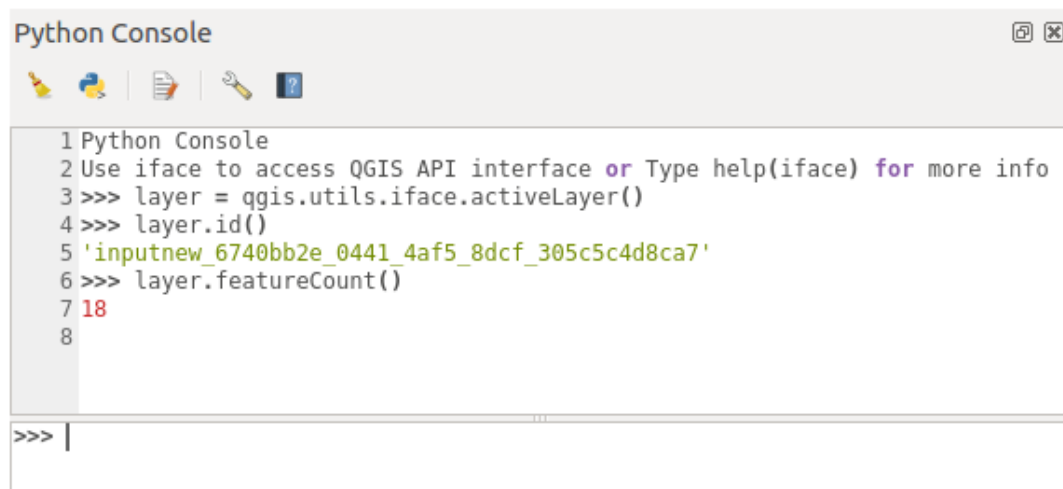
As ligações Python também estão disponíveis para o QGIS Server, incluindo plugins Python (veja *Servidor QGIS e Python*) e ligações Python que podem ser usadas para incorporar o QGIS Server em um aplicativos Python.

Existe uma referência *completa QGIS C++ API* que documenta as classes das bibliotecas QGIS. Uma *API QGIS Pythonica (pyqgis)* é quase idêntica à API C++.

Outro bom recurso para aprender como realizar tarefas comuns é baixar os plugins existentes do “repositório de plugins <<https://plugins.qgis.org/>>” 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*



```
Python Console
1 Python Console
2 Use iface to access QGIS API interface or Type help(iface) for more info
3 >>> layer = qgis.utils.iface.activeLayer()
4 >>> layer.id()
5 'inputnew_6740bb2e_0441_4af5_8dcf_305c5c4d8ca7'
6 >>> layer.featureCount()
7 18
8
>>> |
```

Figura1.1: 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
```

Para aqueles que utilizam o console frequentemente, pode ser útil definir um atalho para acionar o console (dentro de *Configurações ► Atalhos do teclado...*)

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.

1.2.1 Plugins de processamento

Os Plugins de processamento podem ser usados para processar dados. Eles são mais fáceis de desenvolver, mais específicos e mais leves que os Plugins Python. ref:*processing_plugin* explica quando o uso de algoritmos de processamento é apropriado e como desenvolvê-los.

1.3 Executando o código Python quando o QGIS é iniciado

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

1. Criando um script `startup.py`
2. Configurando a variável de ambiente `PYQGIS_STARTUP` em um arquivo Python existente
3. Specifying a startup script using the `--code init_qgis.py` parameter.

1.3.1 O arquivo `startup.py`

Every time QGIS starts, the user's Python home directory and a list of system paths are searched for a file named `startup.py`. If that file exists, it is executed by the embedded Python interpreter.

The path in the user's home directory usually is found under:

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

The default system paths depend on the operating system. To find the paths that work for you, open the Python Console and run `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` to see the list of default directories.

The `startup.py` script is executed immediately upon initializing python in QGIS, early on in the start of the application.

1.3.2 A variável de ambiente `PYQGIS_STARTUP`

Você pode executar código Python logo antes da inicialização do QGIS completando configurando a variável de ambiente `PYQGIS_STARTUP` com o caminho de um arquivo Python existente.

Esse código irá rodar antes da inicialização do QGIS ser completada. Este método é muito útil para a limpeza de `sys.path`, o qual pode haver caminhos indesejáveis, ou para isolar/carregar o ambiente inicial sem a necessidade de um ambiente virtual, ex: instalações homebrew ou MacPorts no Mac.

1.3.3 The `--code` parameter

You can provide custom code to execute as startup parameter to QGIS. To do so, create a python file, for example `qgis_init.py`, to execute and start QGIS from the command line using `qgis --code qgis_init.py`.

Code provided via `--code` is executed late in the QGIS initialization phase, after the application components have been loaded.

1.3.4 Additional arguments for Python

To provide additional arguments for your `--code` script or for other python code that is executed, you can use the `--py-args` argument. Any argument coming after `--py-args` and before a `-- arg` (if present) will be passed to Python but ignored by the QGIS application itself.

In the following example, `myfile.tif` will be available via `sys.argv` in Python but will not be loaded by QGIS. Whereas `otherfile.tif` will be loaded by QGIS but is not present in `sys.argv`.

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

If you want access to every command line parameter from within Python, you can use `QCoreApplication.arguments()`

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

1.4 Aplicações Python

Muitas vezes é útil criar scripts para automatizar processos. Com o PyQGIS, isso é perfeitamente possível — importe o módulo `qgis.core`, inicialize-o e você estará pronto para o processamento.

Ou você pode querer criar aplicações iterativas que utilizem funcionalidades de SIG — realizar medições, exportar um mapa como PDF, ... O módulo `qgis.gui` fornece vários componentes GUI. Mais notavelmente o widget de tela de mapa que pode ser incorporado ao aplicativo com suporte para zoom, panorama e/ou qualquer outra ferramenta de mapa personalizada.

Aplicativos personalizados PyQGIS ou scripts independentes devem ser configurados para localizar os recursos QGIS, como informações de projeção e provedores para leitura de camadas vetoriais e raster. Os recursos QGIS são inicializados adicionando algumas linhas ao início de seu aplicativo ou script. O código para inicializar o QGIS para aplicativos personalizados e scripts independentes é semelhante. Exemplos de cada um são fornecidos abaixo.

Nota: Não utilize `qgis.py` como um nome para seu script. Python não será capaz de importar as ligações, pois o nome do script irá ocultá-las.

1.4.1 Usando PyQGIS em scripts autônomos

Para iniciar um script autônomo, inicialize os recursos do QGIS no início do script:

```
1 from qgis.core import *
2
3 # Supply path to qgis install location
4 QgsApplication.setPrefixPath("/path/to/qgis/installation", True)
5
6 # Create a reference to the QgsApplication. Setting the
7 # second argument to False disables the GUI.
8 qgs = QgsApplication([], False)
9
10 # Load providers
11 qgs.initQgis()
12
13 # Write your code here to load some layers, use processing
14 # algorithms, etc.
15
16 # Finally, exitQgis() is called to remove the
17 # provider and layer registries from memory
18 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.

O caminho de instalação do QGIS varia de acordo com a plataforma; a maneira mais fácil de encontrá-lo para seu sistema é utilizar o *Escrevendo Scripts no Terminal Python* de dentro do QGIS e olhar para a saída da execução:

```
QgsApplication.setPrefixPath()
```

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

```
qgs.initQgis()
```

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

```
qgs.exitQgis()
```

1.4.2 Usando o PyQGIS em aplicativos personalizados

The only difference between *Usando PyQGIS em scripts autônomos* 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.

```

1 from qgis.core import *
2
3 # Supply the path to the qgis install location
4 QgsApplication.setPrefixPath("/path/to/qgis/installation", True)
5
6 # Create a reference to the QgsApplication.
7 # Setting the second argument to True enables the GUI. We need
8 # this since this is a custom application.
9
10 qgs = QgsApplication([], True)
11
12 # load providers
13 qgs.initQgis()
14
15 # Write your code here to load some layers, use processing
16 # algorithms, etc.
17
18 # Finally, exitQgis() is called to remove the
19 # provider and layer registries from memory
20 qgs.exitQgis()
```

Agora você pode trabalhar com o QGIS API - carregar camadas e fazer algum processamento ou acender uma GUI com uma tela de mapa. As possibilidades são infinitas :-)

1.4.3 Executando aplicativos personalizados

Você precisa dizer ao seu sistema onde procurar bibliotecas QGIS e módulos Python apropriados se eles não estiverem em um local bem conhecido - caso contrário, Python reclamará:

```
>>> 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:

- no Linux: **export PYTHONPATH=**`<qgispath>/share/qgis/python`
- no Windows: **set PYTHONPATH=**`c:\<qgispath>\python`
- no 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
```

Conserte isso adicionando os diretórios onde residem as bibliotecas QGIS ao caminho de busca do conector dinâmico:

- no Linux: **export LD_LIBRARY_PATH=**`<qgispath>/lib`
- no Windows: **set PATH=**`C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%` onde `<qgisrelease>` deve ser substituído pelo tipo de liberação que você está visando (por exemplo, `qgis-ltr`, `qgis`, `qgis-dev`)

Estes comandos podem ser colocados em um roteiro de bootstrap que cuidará do início da operação. Ao implantar aplicações personalizadas usando PyQGIS, geralmente há duas possibilidades:

- exigir que o usuário instale o QGIS antes de instalar sua aplicação. O instalador do aplicativo deve procurar por locais padrão das bibliotecas QGIS e permitir que o usuário defina o caminho se não for encontrado. Esta abordagem tem a vantagem de ser mais simples, porém requer que o usuário faça mais etapas.
- pacote QGIS junto com sua aplicação. A liberação do aplicativo pode ser mais desafiadora e o pacote será maior, mas o usuário será salvo do fardo de baixar e instalar partes adicionais de software.

Os dois modelos de implantação podem ser misturados. Você pode fornecer um aplicativo autônomo em Windows e macOS, mas para Linux deixe a instalação de GIS a cargo do usuário e seu gerenciador de pacotes.

1.5 Technical notes on PyQt and SIP

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

Carregando projetos

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

```
1 from qgis.core import (
2     Qgis,
3     QgsProject,
4     QgsPathResolver
5 )
6
7 from qgis.gui import (
8     QgsLayerTreeMapCanvasBridge,
9 )
```

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:

```
1 # If you are not inside a QGIS console you first need to import
2 # qgis and PyQt classes you will use in this script as shown below:
3 from qgis.core import QgsProject
4 # Get the project instance
5 project = QgsProject.instance()
6 # Print the current project file name (might be empty in case no projects have
7 # ↪been loaded)
8 # print(project.fileName())
9
10 # Load another project
11 project.read('testdata/01_project.qgs')
12 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 Resolvendo caminhos errados

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 helps you rewrite layers path within the project.

Its `setPathPreprocessor()` method allows setting a custom path pre-processor function to manipulate 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. If multiple preprocessors are set, they will be called in sequence based on the order in which they were originally set.

Alguns casos de uso:

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)
```

Likewise, a `setPathWriter()` method is available for a path writer function.

Um exemplo para substituir o caminho por uma variável:


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

QgsPathResolver.setPathWriter(my_processor)
```

Both methods return an id that can be used to remove the pre-processor or writer they added. See `removePathPreprocessor()` and `removePathWriter()`.

2.2 Uso de referências para acelerar as coisas

In some instances where you may not need to use a fully functional project, but only want to access it for a specific reason, flags may be helpful. A full list of flags is available under `ProjectReadFlag`. Multiple flags can be added together.

As an example, if we do not care about actual layers and data and simply want to access a project (e.g. for layout or 3D view settings), we can use `DontResolveLayers` flag to bypass the data validation step and prevent the bad layer dialog from appearing. The following can be done:

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

Para acrescentar mais bandeiras, deve-se utilizar o operador Bitwise python OR (`|`).

Carregando Camadas

Dica: 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
)
```

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

3.1 Camadas Vetoriais

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

Para uma camada vetorial geopackage:

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

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

```
vlayer = iface.addVectorLayer(gpkg_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:

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

- for ESRI Shapefile:

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

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

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

- 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:

```
1 uri = QgsDataSourceUri()
2 # set host name, port, database name, username and password
3 uri.setConnection("localhost", "5432", "dbname", "johnny", "xxx")
4 # set database schema, table name, geometry column and optionally
5 # subset (WHERE clause)
6 uri.setDataSource("public", "roads", "the_geom", "cityid = 2643", "primary_key_
  ↳field")
7
8 vlayer = QgsVectorLayer(uri.uri(False), "layer name you like", "postgres")
```

Nota: O argumento `False` 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:

```
1 uri = QgsDataSourceUri()
2 uri.setDatabase('/home/martin/test-2.3.sqlite')
3 schema = ''
4 table = 'Towns'
5 geom_column = 'Geometry'
6 uri.setDataSource(schema, table, geom_column)
7
8 display_name = 'Towns'
9 vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
10 QgsProject.instance().addMapLayer(vlayer)
```

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

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

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

```
uri = "https://demo.mapserver.org/cgi-bin/wfs?service=WFS&version=2.0.0&
↳request=GetFeature&typename=ms:cities"
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")
```

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

```
1 import urllib
2
3 params = {
4     'service': 'WFS',
5     'version': '2.0.0',
6     'request': 'GetFeature',
7     'typename': 'ms:cities',
8     'srsname': "EPSG:4326"
9 }
10 uri2 = 'https://demo.mapserver.org/cgi-bin/wfs?' + urllib.parse.unquote(urllib.
↳parse.urlencode(params))
```

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

```
1 uri = "https://demo.mapserver.org/cgi-bin/wfs?service=WFS&version=2.0.0&
↳request=GetFeature&typename=ms:cities"
2 provider_options = QgsDataProvider.ProviderOptions()
3 # Use project's transform context
4 provider_options.transformContext = QgsProject.instance().transformContext()
5 vlayer.setDataSource(uri, "layer name you like", "WFS", provider_options)
6
7 del(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:

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

Para carregar um raster de um geopackage:

```
1 # get the path to a geopackage e.g. /home/project/data/data.gpkg
2 path_to_gpkg = os.path.join(os.getcwd(), "testdata", "sublayers.gpkg")
3 # gpkg_raster_layer = "GPKG:/home/project/data/data.gpkg:srtm"
4 gpkg_raster_layer = "GPKG:" + path_to_gpkg + ":srtm"
5
6 rlayer = QgsRasterLayer(gpkg_raster_layer, "layer name you like", "gdal")
7
8 if not rlayer.isValid():
9     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.

Para carregar um raster PostGIS:

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

```
1 uri_config = {
2     # database parameters
3     'dbname': 'gis_db',          # The PostgreSQL database to connect to.
4     'host': 'localhost',        # The host IP address or localhost.
5     'port': '5432',             # The port to connect on.
6     'sslmode': QgsDataSourceUri.SslDisable, # SslAllow, SslPrefer, SslRequire,
7     # user and password are not needed if stored in the authcfg or service
8     'authcfg': 'QconfigId',     # The QGIS authentication database ID holding
9     # connection details.
10    'service': None,             # The PostgreSQL service to be used for connection to
11    # the database.
12    'username': None,            # The PostgreSQL user name.
13    'password': None,            # The PostgreSQL password for the user.
14    # table and raster column details
15    'schema': 'public',          # The database schema that the table is located in.
16    'table': 'my_rasters',       # The database table to be loaded.
17    'geometrycolumn': 'rast',    # raster column in PostGIS table
18    'sql': None,                 # An SQL WHERE clause. It should be placed at the end
19    # of the string.
20    'key': None,                 # A key column from the table.
21    'srid': None,                # A string designating the SRID of the coordinate
```

(continua na próxima página)

(continuação da página anterior)

```

19  ↪reference system.
    'estimatedmetadata':'False', # A boolean value telling if the metadata is_
20  ↪estimated.
21  'type':None,                  # A WKT string designating the WKB Type.
22  'selectatid':None,           # Set to True to disable selection by feature ID.
23  'options':None,              # other PostgreSQL connection options not in this list.
24  'enableTime': None,
25  'temporalDefaultTime': None,
26  'temporalFieldIndex': None,
    'mode':'2',                  # GDAL 'mode' parameter, 2 unions raster tiles, 1 adds_
27  ↪tiles separately (may require user input)
28  }
29  # remove any NULL parameters
30  uri_config = {key:val for key, val in uri_config.items() if val is not None}
31  # get the metadata for the raster provider and configure the URI
32  md = QgsProviderRegistry.instance().providerMetadata('postgresraster')
33  uri = QgsDataSourceUri(md.encodeUri(uri_config))
34
35  # the raster can then be loaded into the project
    rlayer = iface.addRasterLayer(uri.uri(False), "raster layer name", "postgresraster"
    ↪")

```

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

```

layer_name = 'modis'
url = "https://demo.mapserver.org/cgi-bin/wcs?identifier={}".format(layer_name)
rlayer = QgsRasterLayer(url, '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=continents&styles&
↪url=https://demo.mapserver.org/cgi-bin/wms"
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
    print("Layer failed to load!")
```

3.3 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:

```
1 # first add the layer without showing it
2 QgsProject.instance().addMapLayer(rlayer, False)
3 # obtain the layer tree of the top-level group in the project
4 layerTree = iface.layerTreeCanvasBridge().rootGroup()
5 # the position is a number starting from 0, with -1 an alias for the end
6 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()
```

Acessando o sumário (TOC)

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

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

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': <QgsVectorLayer: '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:

```
1 # list of layer names using list comprehension
2 l = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
```

(continua na próxima página)

(continuação da página anterior)

```
3 # dictionary with key = layer name and value = layer object
4 layers_list = {}
5 for l in QgsProject.instance().mapLayers().values():
6     layers_list[l.name()] = l
7
8 print(layers_list)
```

```
{'countries': <QgsVectorLayer: '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: for more information you can read these blog posts of Martin Dobias: [Part 1](#) [Part 2](#) [Part 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)
```

```
<QgsLayerTreeLayer: countries>
```

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)
```

```
[<QgsVectorLayer: '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()`:

```
1 # create a temporary layer
2 layer1 = QgsVectorLayer("path_to_layer", "Layer 1", "memory")
3 # add the layer to the legend, last position
4 root.addLayer(layer1)
5 # add the layer at given position
6 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: <QgsLayerTreeLayer: countries>
Map layer: <QgsVectorLayer: '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

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

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

```
1 # get a QgsVectorLayer
2 vl = QgsProject.instance().mapLayersByName("countries")[0]
3 # create a QgsLayerTreeLayer object from vl by its id
4 myvl = root.findLayer(vl.id())
5 # clone the myvl QgsLayerTreeLayer object
```

(continua na próxima página)

(continuação da página anterior)

```
6 myvlclone = myvl.clone()
7 # get the parent. If None (layer is not in group) returns ''
8 parent = myvl.parent()
9 # move the cloned layer to the top (0)
10 parent.insertChildNode(0, myvlclone)
11 # remove the original myvl
12 root.removeChildNode(myvl)
```

ou movê-la para um grupo existente:

```
1 # get a QgsVectorLayer
2 vl = QgsProject.instance().mapLayersByName("countries")[0]
3 # create a QgsLayerTreeLayer object from vl by its id
4 myvl = root.findLayer(vl.id())
5 # clone the myvl QgsLayerTreeLayer object
6 myvlclone = myvl.clone()
7 # create a new group
8 group1 = root.addGroup("Group1")
9 # get the parent. If None (layer is not in group) returns ''
10 parent = myvl.parent()
11 # move the cloned layer to the top (0)
12 group1.insertChildNode(0, myvlclone)
13 # remove the QgsLayerTreeLayer from its parent
14 parent.removeChildNode(myvl)
```

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

```
1 node_group1 = root.findGroup("Group1")
2 # change the name of the group
3 node_group1.setName("Group X")
4 node_layer2 = root.findLayer(country_layer.id())
5 # change the name of the layer
6 node_layer2.setName("Layer X")
7 # change the visibility of a layer
8 node_group1.setItemVisibilityChecked(True)
9 node_layer2.setItemVisibilityChecked(False)
10 # expand/collapse the group view
11 node_group1.setExpanded(True)
12 node_group1.setExpanded(False)
```

Usando Camadas Raster

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

```
1 from qgis.core import (
2     QgsRasterLayer,
3     QgsProject,
4     QgsPointXY,
5     QgsRaster,
6     QgsRasterShader,
7     QgsColorRampShader,
8     QgsSingleBandPseudoColorRenderer,
9     QgsSingleBandColorDataRenderer,
10    QgsSingleBandGrayRenderer,
11 )
12
13 from qgis.PyQt.QtGui import (
14     QColor,
15 )
```

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 -
↳33.750775007000000144>
```

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

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

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

```
0
```

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

```
1
```

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

```
Band 1: Height
```

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

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

5.2 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, utilize o método `setRenderer()` método de `QgsRasterLayer`. Há um número de classes de renderizador (derivado de `QgsRasterRenderer`):

- `QgsHillshadeRenderer`
- `QgsMultiBandColorRenderer`

- `QgsPalettedRasterRenderer`
- `QgsRasterContourRenderer`
- `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:

```
1 fcn = QgsColorRampShader()
2 fcn.setColorRampType(QgsColorRampShader.Interpolated)
3 lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)),
4         QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
5 fcn.setColorRampItemList(lst)
6 shader = QgsRasterShader()
7 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 utilizar 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 assim criar uma imagem colorida (esse é o estilo de desenho “MultiBandColor”). Em alguns casos, você pode querer substituir essas configurações. O código a seguir intercala a faixa vermelha (1) e a 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 utilizar `triggerRepaint()` para atualizar o mapa e ver o resultado:

```
rlayer_multi.triggerRepaint()
```

5.3 Valores de Consulta

Valores Raster podem ser consultados utilizando o método `sample()` método da `QgsRasterDataProvider` class. Você tem que especificar uma `QgsPointXY` e o número da banda da camada raster que você deseja consultar. O método retorna um tuplo com o valor e Verdadeiro ou Falso, dependendo dos resultados:

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

Outra forma de 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}"

5.4 Edição de dados raster

Você pode criar uma camada raster utilizando a `QgsRasterBlock` class. Por exemplo, para criar um bloco raster 2x2 com um byte por pixel:

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

Os pixels raster podem ser sobrescritos graças ao método `writeBlock()`. Para sobrescrever dados raster existentes na posição 0,0 pelo bloco 2x2:

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

Usando Camadas Vetoriais

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

```
1 from qgis.core import (
2     QgsApplication,
3     QgsDataSourceUri,
4     QgsCategorizedSymbolRenderer,
5     QgsClassificationRange,
6     QgsPointXY,
7     QgsProject,
8     QgsExpression,
9     QgsField,
10    QgsFields,
11    QgsFeature,
12    QgsFeatureRequest,
13    QgsFeatureRenderer,
14    QgsGeometry,
15    QgsGraduatedSymbolRenderer,
16    QgsMarkerSymbol,
17    QgsMessageLog,
18    QgsRectangle,
19    QgsRendererCategory,
20    QgsRendererRange,
21    QgsSymbol,
22    QgsVectorDataProvider,
23    QgsVectorLayer,
24    QgsVectorFileWriter,
25    QgsWkbTypes,
26    QgsSpatialIndex,
27    QgsVectorLayerUtils
28 )
29
30 from qgis.core.additions.edit import edit
31
32 from qgis.PyQt.QtGui import (
33     QColor,
34 )
```

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

```
1 fid Integer64
2 id Integer64
3 scalerank Integer64
4 featurecla String
5 type String
6 name String
7 abbrev String
8 location String
9 gps_code String
10 iata_code String
11 wikipedia String
12 natlscale Real
```

The `displayField()` and `mapTipTemplate()` methods 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/data/data.gpkg|layername=airports", "Airports_
↪layer", "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`.

```
1 # "layer" is a QgsVectorLayer instance
2 layer = iface.activeLayer()
3 features = layer.getFeatures()
4
5 for feature in features:
6     # retrieve every feature with its geometry and attributes
7     print("Feature ID: ", feature.id())
```

(continua na próxima página)

(continuação da página anterior)

```

8  # fetch geometry
9  # show some information about the feature geometry
10 geom = feature.geometry()
11 geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
12 if geom.type() == QgsWkbTypes.PointGeometry:
13     # the geometry type can be of single or multi type
14     if geomSingleType:
15         x = geom.asPoint()
16         print("Point: ", x)
17     else:
18         x = geom.asMultiPoint()
19         print("MultiPoint: ", x)
20 elif geom.type() == QgsWkbTypes.LineGeometry:
21     if geomSingleType:
22         x = geom.asPolyline()
23         print("Line: ", x, "length: ", geom.length())
24     else:
25         x = geom.asMultiPolyline()
26         print("MultiLine: ", x, "length: ", geom.length())
27 elif geom.type() == QgsWkbTypes.PolygonGeometry:
28     if geomSingleType:
29         x = geom.asPolygon()
30         print("Polygon: ", x, "Area: ", geom.area())
31     else:
32         x = geom.asMultiPolygon()
33         print("MultiPolygon: ", x, "Area: ", geom.area())
34 else:
35     print("Unknown or invalid geometry")
36 # fetch attributes
37 attrs = feature.attributes()
38 # attrs is a list. It contains all the attribute values of this feature
39 print(attrs)
40 # for this test only print the first feature
41 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()

```

(continua na próxima página)

(continuação da página anterior)

```
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:

```
1 selected_fid = []
2
3 # Get the first feature id from the layer
4 feature = next(layer.getFeatures())
5 if feature:
6     selected_fid.append(feature.id())
7
8 # Add that features to the selected list
9 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:

```
1 areaOfInterest = QgsRectangle(450290,400520, 450750,400780)
2
3 request = QgsFeatureRequest().setFilterRect(areaOfInterest)
4
5 for feature in layer.getFeatures(request):
6     # do whatever you need with the feature
7     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>
<qgis._core.QgsFeature object at 0x7faef5881670>
```

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.

```
1 # Only return selected fields to increase the "speed" of the request
2 request.setSubsetOfAttributes([0,2])
3
4 # More user friendly version
5 request.setSubsetOfAttributes(['name','id'],layer.fields())
6
7 # Don't return geometry objects to increase the "speed" of the request
8 request.setFlags(QgsFeatureRequest.NoGeometry)
9
10 # Fetch only the feature with id 45
11 request.setFilterFid(45)
12
13 # The options may be chained
14 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:

```
1 caps_string = layer.dataProvider().capabilitiesString()
2 # Print:
3 # 'Add Features, Delete Features, Change Attribute Values, Add Attributes,
4 # Delete Attributes, Rename Attributes, Fast Access to Features at ID,
5 # Presimplify Geometries, Presimplify Geometries with Validity Check,
6 # 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:

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

6.4.1 Adicionar feições

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

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.

```
1 if caps & QgsVectorDataProvider.AddFeatures:
2     feat = QgsFeature(layer.fields())
3     feat.setAttributes([0, 'hello'])
4     # Or set a single attribute by key or by index:
5     feat.setAttribute('name', 'hello')
6     feat.setAttribute(0, 'hello')
7     feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(123, 456)))
8     (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.

```
1 fid = 100    # ID of the feature we will modify
2
3 if caps & QgsVectorDataProvider.ChangeAttributeValues:
4     attrs = { 0 : "hello", 1 : 123 }
5     layer.dataProvider().changeAttributeValues({ fid : attrs })
6
7 if caps & QgsVectorDataProvider.ChangeGeometries:
8     geom = QgsGeometry.fromPointXY(QgsPointXY(111,222))
9     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.

```
1 from qgis.PyQt.QtCore import QMetaType
2
3 feat1 = feat2 = QgsFeature(layer.fields())
4 fid = 99
5 feat1.setId(fid)
6
7 # add two features (QgsFeature instances)
8 layer.addFeatures([feat1, feat2])
9 # delete a feature with specified ID
10 layer.deleteFeature(fid)
```

(continua na próxima página)

(continuação da página anterior)

```

11
12 # set new geometry (QgsGeometry instance) for a feature
13 geometry = QgsGeometry.fromWkt("POINT(7 45)")
14 layer.changeGeometry(fid, geometry)
15 # update an attribute with given field index (int) to a given value
16 fieldIndex = 1
17 value = 'My new name'
18 layer.changeAttributeValue(fid, fieldIndex, value)
19
20 # add new field
21 layer.addAttribute(QgsField("mytext", QMetaType.Type.QString))
22 # remove a field
23 layer.deleteAttribute(fieldIndex)

```

Para desfazer/refazer o trabalho corretamente, os chamados acima mencionados têm de ser envolvidos 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:

```

1 layer.beginEditCommand("Feature triangulation")
2
3 # ... call layer's editing methods ...
4
5 if problem_occurred:
6     layer.destroyEditCommand()
7     # ... tell the user that there was a problem
8     # and return
9
10 # ... more editing ...
11
12 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.

```

1 from qgis.PyQt.QtCore import QMetaType
2
3 if caps & QgsVectorDataProvider.AddAttributes:
4     res = layer.dataProvider().addAttributes(
5         [QgsField("mytext", QMetaType.Type.QString),
6           QgsField("myint", QMetaType.Type.Int)])
7
8 if caps & QgsVectorDataProvider.DeleteAttributes:
9     res = layer.dataProvider().deleteAttributes([0])

```

```

1 # Alternate methods for removing fields
2 # first create temporary fields to be removed (f1-3)
3 layer.dataProvider().addAttributes([QgsField("f1", QMetaType.Type.Int),
4                                     QgsField("f2", QMetaType.Type.Int),
5                                     QgsField("f3", QMetaType.Type.Int)])
6 layer.updateFields()
7 count=layer.fields().count() # count of layer fields
8 ind_list=list((count-3, count-2)) # create list
9
10 # remove a single field with an index
11 layer.dataProvider().deleteAttributes([count-1])
12
13 # remove multiple fields with a list of indices
14 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:

- create spatial index using the `QgsSpatialIndex` class:

```
index = QgsSpatialIndex()
```

- adicione feições ao índice — o índice 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(feats)
```

- 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

```
1 # returns array of feature IDs of five nearest features
2 nearest = index.nearestNeighbor(QgsPointXY(25.4, 12.7), 5)
3
4 # returns array of IDs of features which intersect the rectangle
5 intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
```

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

- suporta **apenas** feições de ponto único
- is **static** (no additional features can be added to the index after the construction)
- **é muito mais rápido!**
- allows direct retrieval of the original feature's points, without requiring additional feature requests
- supports true *distance based* searches, i.e. return all points within a radius from a search point

6.6 The `QgsVectorLayerUtils` class

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

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

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

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

```
1 vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports_
↳layer", "ogr")
2 # select only the first feature to make the output shorter
3 vlayer.selectByIds([1])
4 val = QgsVectorLayerUtils.getValues(vlayer, "NAME", selectedOnly=True)
5 print(val)
```

```
(['Sahnewal'], True)
```

6.7 Criando Camadas Vetoriais

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

- the `QgsVectorFileWriter` class: A convenient class for writing vector files to disk, using either a static call to `writeAsVectorFormatV3()` which saves the whole vector layer or creating an instance of the class and issue calls to inherited `addFeature()`. This class supports all the vector formats that GDAL supports (GeoPackage, Shapefile, GeoJSON, KML and others).
- the `QgsVectorLayer` class: instantiates a data provider that interprets the supplied path (url) of the data source to connect to and access the data. It can be used to create temporary, memory-based layers (memory) and connect to GDAL vector datasets (ogr), databases (postgres, spatialite, mysql, mssql) and more (wfs, gpx, delimitedtext...).

6.7.1 De uma instância de `QgsVectorFileWriter`

```

1  # SaveVectorOptions contains many settings for the writer process
2  save_options = QgsVectorFileWriter.SaveVectorOptions()
3  transform_context = QgsProject.instance().transformContext()
4  # Write to a GeoPackage (default)
5  error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
6                                                    "testdata/my_new_file.gpkg",
7                                                    transform_context,
8                                                    save_options)
9  if error[0] == QgsVectorFileWriter.NoError:
10     print("success!")
11  else:
12     print(error)

```

```

1  # Write to an ESRI Shapefile format dataset using UTF-8 text encoding
2  save_options = QgsVectorFileWriter.SaveVectorOptions()
3  save_options.driverName = "ESRI Shapefile"
4  save_options.fileEncoding = "UTF-8"
5  transform_context = QgsProject.instance().transformContext()
6  error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
7                                                    "testdata/my_new_shapefile",
8                                                    transform_context,
9                                                    save_options)
10  if error[0] == QgsVectorFileWriter.NoError:
11     print("success again!")
12  else:
13     print(error)

```

```

1  # Write to an ESRI GDB file
2  save_options = QgsVectorFileWriter.SaveVectorOptions()
3  save_options.driverName = "FileGDB"
4  # if no geometry
5  save_options.overrideGeometryType = QgsWkbTypes.Unknown
6  save_options.actionOnExistingFile = QgsVectorFileWriter.CreateOrOverwriteLayer
7  save_options.layerName = 'my_new_layer_name'
8  transform_context = QgsProject.instance().transformContext()
9  gdb_path = "testdata/my_example.gdb"
10  error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
11                                                    gdb_path,
12                                                    transform_context,
13                                                    save_options)
14  if error[0] == QgsVectorFileWriter.NoError:
15     print("success!")
16  else:
17     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:

```

1 LIST_FIELD_NAME = 'xxxx'
2
3 class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):
4
5     def __init__(self, layer, list_field):
6         QgsVectorFileWriter.FieldValueConverter.__init__(self)
7         self.layer = layer
8         self.list_field_idx = self.layer.fields().indexOfName(list_field)
9
10    def convert(self, fieldIdxInLayer, value):
11        if fieldIdxInLayer == self.list_field_idx:
12            return QgsListFieldFormatter().representValue(layer=vlayer,
13                                                         fieldIndex=self.list_field_idx,
14                                                         config={},
15                                                         cache=None,
16                                                         value=value)
17
18        else:
19            return value
20
21    def fieldDefinition(self, field):
22        idx = self.layer.fields().indexOfName(field.name())
23        if idx == self.list_field_idx:
24            return QgsField(LIST_FIELD_NAME, QMetaType.Type.QString)
25        else:
26            return self.layer.fields()[idx]
27
28 converter = ESRIValueConverter(vlayer, LIST_FIELD_NAME)
29 opts = QgsVectorFileWriter.SaveVectorOptions()
30 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.

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

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.

6.7.2 Diretamente das feições

```

1 from qgis.PyQt.QtCore import QMetaType
2
3 # define fields for feature attributes. A QgsFields object is needed
4 fields = QgsFields()
5 fields.append(QgsField("first", QMetaType.Type.Int))
6 fields.append(QgsField("second", QMetaType.Type.QString))
7
8 """ create an instance of vector file writer, which will create the vector file.
9 Arguments:
10 1. path to new file (will fail if exists already)
11 2. field map
12 3. geometry type - from WKBTYP enum
13 4. layer's spatial reference (instance of
14    QgsCoordinateReferenceSystem)
15 5. coordinate transform context

```

(continua na próxima página)

(continuação da página anterior)

```

16 6. save options (driver name for the output file, encoding etc.)
17 """
18
19 crs = QgsProject.instance().crs()
20 transform_context = QgsProject.instance().transformContext()
21 save_options = QgsVectorFileWriter.SaveVectorOptions()
22 save_options.driverName = "ESRI Shapefile"
23 save_options.fileEncoding = "UTF-8"
24
25 writer = QgsVectorFileWriter.create(
26     "testdata/my_new_shapefile.shp",
27     fields,
28     QgsWkbTypes.Point,
29     crs,
30     transform_context,
31     save_options
32 )
33
34 if writer.hasError() != QgsVectorFileWriter.NoError:
35     print("Error when creating shapefile: ", writer.errorMessage())
36
37 # add a feature
38 fet = QgsFeature()
39
40 fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
41 fet.setAttributes([1, "text"])
42 writer.addFeature(fet)
43
44 # delete the writer to flush features to disk
45 del writer

```

6.7.3 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

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

index=yes

Especifica que o provedor irá usar o index espacial

field=name:type(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

```
1 from qgis.PyQt.QtCore import QMetaType
2
3 # create layer
4 vl = QgsVectorLayer("Point", "temporary_points", "memory")
5 pr = vl.dataProvider()
6
7 # add fields
8 pr.addAttributes([QgsField("name", QMetaType.Type.QString),
9                     QgsField("age", QMetaType.Type.Int),
10                    QgsField("size", QMetaType.Type.Double)])
11 vl.updateFields() # tell the vector layer to fetch changes from the provider
12
13 # add a feature
14 fet = QgsFeature()
15 fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
16 fet.setAttributes(["Johnny", 2, 0.3])
17 pr.addFeatures([fet])
18
19 # update layer's extent when new features have been added
20 # because change of extent in provider is not propagated to the layer
21 vl.updateExtents()
```

Finalmente, vamos verificar se tudo correu bem

```
1 # show some stats
2 print("fields:", len(pr.fields()))
3 print("features:", pr.featureCount())
4 e = vl.extent()
5 print("extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum())
6
7 # iterate over features
8 features = vl.getFeatures()
9 for fet in features:
10     print("F:", fet.id(), fet.attributes(), fet.geometry().asPoint())
```

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

6.8 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	<code>QgsSingleSymbolRenderer</code>	Renderiza todas as características com o mesmo símbolo
categorizedSymbol	<code>QgsCategorizedSymbolRenderer</code>	Renderiza características usando um símbolo diferente para cada categoria
graduatedSymbol	<code>QgsGraduatedSymbolRenderer</code>	Renderiza características usando diferentes 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', 'mergedFeatureRenderer',
→ 'invertedPolygonRenderer', 'heatmapRenderer', '25dRenderer', 'embeddedSymbol']
```

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
```

6.8.1 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', 'cap_style': 'square', 'color': '255,0,0,255,rgb:1,0,0,1',
↪ 'horizontal_anchor_point': '1', 'joinstyle': 'bevel', 'name': 'square', 'offset'
↪ ': '0,0', 'offset_map_unit_scale': '3x:0,0,0,0,0,0', 'offset_unit': 'MM',
↪ 'outline_color': '35,35,35,255,rgb:0.13725490196078433,0.13725490196078433,0.
↪ 13725490196078433,1', 'outline_style': 'solid', 'outline_width': '0', 'outline_
↪ width_map_unit_scale': '3x:0,0,0,0,0,0', 'outline_width_unit': 'MM', 'scale_
↪ method': 'diameter', 'size': '2', 'size_map_unit_scale': '3x:0,0,0,0,0,0', 'size_
↪ unit': 'MM', 'vertical_anchor_point': '1'}
```

Isso pode ser útil se você quiser alterar algumas propriedades:

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

6.8.2 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

```
1 categorized_renderer = QgsCategorizedSymbolRenderer()
2 # Add a few categories
3 cat1 = QgsRendererCategory('1', QgsMarkerSymbol(), 'category 1')
4 cat2 = QgsRendererCategory('2', QgsMarkerSymbol(), 'category 2')
5 categorized_renderer.addCategory(cat1)
6 categorized_renderer.addCategory(cat2)
7
8 for cat in categorized_renderer.categories():
9     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()`.

6.8.3 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

```
1 graduated_renderer = QgsGraduatedSymbolRenderer()
2 # Add a few categories
3 graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class 0-
   ↳100', 0, 100), QgsMarkerSymbol()))
4 graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class
   ↳101-200', 101, 200), QgsMarkerSymbol()))
5
6 for ran in graduated_renderer.ranges():
7     print("{} - {}: {} {}".format(
8         ran.lowerValue(),
9         ran.upperValue(),
10        ran.label(),
11        ran.symbol()
12    ))
```

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

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

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)

```
1 from qgis.PyQt import QtGui
2
3 myVectorLayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports",
   ↳"Airports layer", "ogr")
4 myTargetField = 'scalerank'
5 myRangeList = []
6 myOpacity = 1
7 # Make our first symbol and range...
8 myMin = 0.0
9 myMax = 50.0
10 myLabel = 'Group 1'
11 myColour = QtGui.QColor('#ffee00')
12 mySymbol1 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
13 mySymbol1.setColor(myColour)
14 mySymbol1.setOpacity(myOpacity)
15 myRange1 = QgsRendererRange(myMin, myMax, mySymbol1, myLabel)
16 myRangeList.append(myRange1)
17 #now make another symbol and range...
```

(continua na próxima página)

```

18 myMin = 50.1
19 myMax = 100
20 myLabel = 'Group 2'
21 myColour = QtGui.QColor('#00eeff')
22 mySymbol2 = QgsSymbol.defaultSymbol(
23     myVectorLayer.geometryType())
24 mySymbol2.setColor(myColour)
25 mySymbol2.setOpacity(myOpacity)
26 myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
27 myRangeList.append(myRange2)
28 myRenderer = QgsGraduatedSymbolRenderer(' ', myRangeList)
29 myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
    ↪ "EqualInterval")
30 myRenderer.setClassificationMethod(myClassificationMethod)
31 myRenderer.setClassAttribute(myTargetField)
32
33 myVectorLayer.setRenderer(myRenderer)

```

6.8.4 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.

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

```

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

```

```
0: SimpleMarker
```

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

Por padrão, tamanho e largura são em milímetros 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 feições 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:

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

```
1 AnimatedMarker
2 EllipseMarker
3 FilledMarker
4 FontMarker
5 GeometryGenerator
6 MaskMarker
7 RasterMarker
8 SimpleMarker
9 SvgMarker
10 VectorField
```

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

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

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

```
1 from qgis.core import QgsMarkerSymbolLayer
2 from qgis.PyQt.QtGui import QColor
3
4 class FooSymbolLayer(QgsMarkerSymbolLayer):
5
6     def __init__(self, radius=4.0):
7         QgsMarkerSymbolLayer.__init__(self)
8         self.radius = radius
9         self.color = QColor(255,0,0)
10
11     def layerType(self):
12         return "FooMarker"
13
14     def properties(self):
15         return { "radius" : str(self.radius) }
16
17     def startRender(self, context):
18         pass
```

(continua na próxima página)

```

19
20 def stopRender(self, context):
21     pass
22
23 def renderPoint(self, point, context):
24     # Rendering depends on whether the symbol is selected (QGIS >= 1.5)
25     color = context.selectionColor() if context.selected() else self.color
26     p = context.renderContext().painter()
27     p.setPen(color)
28     p.drawEllipse(point, self.radius, self.radius)
29
30 def clone(self):
31     return FooSymbolLayer(self.radius)

```

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

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

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

```

1 from qgis.gui import QgsSymbolLayerWidget
2
3 class FooSymbolLayerWidget(QgsSymbolLayerWidget):
4     def __init__(self, parent=None):
5         QgsSymbolLayerWidget.__init__(self, parent)
6
7         self.layer = None
8
9         # setup a simple UI
10        self.label = QLabel("Radius:")
11        self.spinRadius = QDoubleSpinBox()
12        self.hbox = QHBoxLayout()
13        self.hbox.addWidget(self.label)
14        self.hbox.addWidget(self.spinRadius)
15        self.setLayout(self.hbox)
16        self.connect(self.spinRadius, SIGNAL("valueChanged(double)"), \
17                    self.radiusChanged)
18
19        def setSymbolLayer(self, layer):
20            if layer.layerType() != "FooMarker":
21                return
22            self.layer = layer
23            self.spinRadius.setValue(layer.radius)
24
25        def symbolLayer(self):
26            return self.layer
27
28        def radiusChanged(self, value):
29            self.layer.radius = value
30            self.emit(SIGNAL("changed()"))

```

This widget can be embedded into the symbol properties dialog. When the symbol layer type is selected in symbol properties dialog, it creates an instance of the symbol layer and an instance of the symbol layer widget. Then it calls the `setSymbolLayer()` method to assign the symbol layer to the widget. In that method the widget should update

the UI to reflect the attributes of the symbol layer. The `symbolLayer()` method is used to retrieve the symbol layer again by the properties dialog to use it for the symbol.

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

```
1 from qgis.core import QgsSymbol, QgsSymbolLayerAbstractMetadata, \
   ↪QgsSymbolLayerRegistry
2
3 class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):
4
5     def __init__(self):
6         super().__init__("FooMarker", "My new Foo marker", QgsSymbol.Marker)
7
8     def createSymbolLayer(self, props):
9         radius = float(props["radius"]) if "radius" in props else 4.0
10        return FooSymbolLayer(radius)
11
12 fslmetadata = FooSymbolLayerMetadata()
```

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

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.

6.8.5 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

```
1 import random
2 from qgis.core import QgsWkbTypes, QgsSymbol, QgsFeatureRenderer
3
4
5 class RandomRenderer(QgsFeatureRenderer):
6     def __init__(self, syms=None):
7         super().__init__("RandomRenderer")
8         self.syms = syms if syms else [
9             QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point)),
10            QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point))
11        ]
12
13     def symbolForFeature(self, feature, context):
14         return random.choice(self.syms)
15
16     def startRender(self, context, fields):
17         super().startRender(context, fields)
```

(continua na próxima página)

(continuação da página anterior)

```

18     for s in self.syms:
19         s.startRender(context, fields)
20
21     def stopRender(self, context):
22         super().stopRender(context)
23         for s in self.syms:
24             s.stopRender(context)
25
26     def usedAttributes(self, context):
27         return []
28
29     def clone(self):
30         return RandomRendererer(self.syms)

```

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

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

```

1  from qgis.gui import QgsRendererWidget, QgsColorButton
2
3
4  class RandomRendererWidget(QgsRendererWidget):
5      def __init__(self, layer, style, renderer):
6          super().__init__(layer, style)
7          if renderer is None or renderer.type() != "RandomRenderer":
8              self.r = RandomRenderer()
9          else:
10             self.r = renderer
11             # setup UI
12             self.btn1 = QgsColorButton()
13             self.btn1.setColor(self.r.syms[0].color())
14             self.vbox = QVBoxLayout()
15             self.vbox.addWidget(self.btn1)
16             self.setLayout(self.vbox)
17             self.btn1.colorChanged.connect(self.setColor1)
18
19     def setColor1(self):
20         color = self.btn1.color()
21         if not color.isValid(): return
22         self.r.syms[0].setColor(color)
23
24     def renderer(self):
25         return self.r

```

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

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`

```

1 from qgis.core import (
2     QgsRendererAbstractMetadata,
3     QgsRendererRegistry,
4     QgsApplication
5 )
6
7 class RandomRendererMetadata(QgsRendererAbstractMetadata):
8
9     def __init__(self):
10         super().__init__("RandomRenderer", "Random renderer")
11
12     def createRenderer(self, element):
13         return RandomRenderer()
14
15     def createRendererWidget(self, layer, style, renderer):
16         return RandomRendererWidget(layer, style, renderer)
17
18 rmetadata = RandomRendererMetadata()

```

```
QgsApplication.rendererRegistry().addRenderer(rmetadata)
```

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

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")))

```

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

6.9 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

Manipulação Geométrica

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

```
1 from qgis.core import (  
2     QgsGeometry,  
3     QgsGeometryCollection,  
4     QgsPoint,  
5     QgsPointXY,  
6     QgsWkbTypes,  
7     QgsProject,  
8     QgsFeatureRequest,  
9     QgsVectorLayer,  
10    QgsDistanceArea,  
11    QgsUnitTypes,  
12    QgsCoordinateTransform,  
13    QgsCoordinateReferenceSystem  
14 )
```

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

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

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

7.1 Construção de Geometria

O PyQGIS fornece várias opções para criar uma geometria:

- a partir das coordenadas

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

As coordenadas são dadas usando a classe `QgsPoint` ou a classe `QgsPointXY`. A diferença 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, formas 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)

```
1 g = QgsGeometry()
2 wkb = bytes.fromhex("01010000000000000000000045400000000000001440")
3 g.fromWkb(wkb)
4
5 # print WKT representation of the geometry
6 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.

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

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

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

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

```

1 print(QgsWkbTypes.displayString(gPnt.wkbType()))
2 # output: 'Point'
3 print(QgsWkbTypes.displayString(gLine.wkbType()))
4 # output: 'LineString'
5 print(QgsWkbTypes.displayString(gPolygon.wkbType()))
6 # output: 'Polygon'

```

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

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

```

1 print(gPnt.asPoint())
2 # output: <QgsPointXY: POINT(1 1)>
3 print(gLine.asPolyline())
4 # output: [<QgsPointXY: POINT(1 1)>, <QgsPointXY: POINT(2 2)>]
5 print(gPolygon.asPolygon())
6 # 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()`.

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

```

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

```

```

Point (0 0)
Point (1 1)
Point (2 2)

```

```

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

```

```

LineString (0 0, 10 10)

```

```

gc = QgsGeometryCollection()
gc.fromWkt( 'GeometryCollection( Point(1 2), Point(11 12), LineString(33 34, 44 45))
  ↳' )
print(gc[1].asWkt())

```

```

Point (11 12)

```

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

```

1 geom = QgsGeometry.fromWkt( 'MultiPoint( 0 0, 1 1, 2 2)' )
2 for part in geom.parts():
3     part.transform(QgsCoordinateTransform(
4         QgsCoordinateReferenceSystem("EPSG:4326"),
5         QgsCoordinateReferenceSystem("EPSG:3111"),
6         QgsProject.instance())
7     )

```

(continua na próxima página)

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

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

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.

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

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

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.

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

(continua na próxima página)

(continuação da página anterior)

```

11 geom = f.geometry()
12 name = f.attribute('NAME')
13 print(name)
14 print("Perimeter (m):", d.measurePerimeter(geom))
15 print("Area (m2):", d.measureArea(geom))
16
17 # let's calculate and print the area again, but this time in square kilometers
18 print("Area (km2):", d.convertAreaMeasurement(d.measureArea(geom), QgsUnitTypes.
↳ AreaSquareKilometers))

```

```

1 Zambia
2 Perimeter (m): 5539361.250294601
3 Area (m2): 751989035032.9031
4 Area (km2): 751989.0350329031
5 Zimbabwe
6 Perimeter (m): 2865021.3325076113
7 Area (m2): 389267821381.6008
8 Area (km2): 389267.8213816008

```

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

```

1 d = QgsDistanceArea()
2 d.setEllipsoid('WGS84')
3
4 # Let's create two points.
5 # Santa claus is a workaholic and needs a summer break,
6 # lets see how far is Tenerife from his home
7 santa = QgsPointXY(25.847899, 66.543456)
8 tenerife = QgsPointXY(-16.5735, 28.0443)
9
10 print("Distance in meters: ", d.measureLine(santa, tenerife))

```

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

- Distance and area using the `QgsDistanceArea` class: [Distance matrix algorithm](#)
- [Lines to polygons algorithm](#)

Suporte a projeções

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

```
1 from qgis.core import (  
2     QgsCoordinateReferenceSystem,  
3     QgsCoordinateTransform,  
4     QgsProject,  
5     QgsPointXY,  
6 )
```

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")  
print(crs.isValid())
```

```
True
```

O QGIS suporta identificadores SRC diferentes com os seguintes formatos:

- EPSG:<code> — 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)

```
1 wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.
   ↪257223563]],' \
2     'PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],' \
3     'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
4 crs = QgsCoordinateReferenceSystem(wkt)
5 print(crs.isValid())
```

```
True
```

- 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")
print(crs.isValid())
```

```
True
```

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

```
1 crs = QgsCoordinateReferenceSystem("EPSG:4326")
2
3 print("QGIS CRS ID:", crs.srsid())
4 print("PostGIS SRID:", crs.postgisSrid())
5 print("Description:", crs.description())
6 print("Projection Acronym:", crs.projectionAcronym())
7 print("Ellipsoid Acronym:", crs.ellipsoidAcronym())
8 print("Proj String:", crs.toProj())
9 # check whether it's geographic or projected coordinate system
10 print("Is geographic:", crs.isGeographic())
11 # check type of map units in this CRS (values defined in Qgis::units enum)
12 print("Map units:", crs.mapUnits())
```

Saída:

```
1 QGIS CRS ID: 3452
2 PostGIS SRID: 4326
3 Description: WGS 84
4 Projection Acronym: longlat
5 Ellipsoid Acronym: EPSG:7030
6 Proj String: +proj=longlat +datum=WGS84 +no_defs
7 Is geographic: True
8 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.

```
1 crsSrc = QgsCoordinateReferenceSystem("EPSG:4326")      # WGS 84
2 crsDest = QgsCoordinateReferenceSystem("EPSG:32633")    # WGS 84 / UTM zone 33N
3 transformContext = QgsProject.instance().transformContext()
4 xform = QgsCoordinateTransform(crsSrc, crsDest, transformContext)
5
6 # forward transformation: src -> dest
7 pt1 = xform.transform(QgsPointXY(18,5))
8 print("Transformed point:", pt1)
9
10 # inverse transformation: dest -> src
11 pt2 = xform.transform(pt1, QgsCoordinateTransform.ReverseTransform)
12 print("Transformed back:", pt2)
```

Saída:

```
Transformed point: <QgsPointXY: POINT(832713.79873844375833869 553423.
↪98688333143945783)>
Transformed back: <QgsPointXY: POINT(18 4.99999999999999911)>
```

Usando a Tela do Mapa

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

```
1 from qgis.PyQt.QtGui import (
2     QColor,
3 )
4
5 from qgis.PyQt.QtCore import Qt, QRectF
6
7 from qgis.PyQt.QtWidgets import QMenu
8
9 from qgis.core import (
10     QgsVectorLayer,
11     QgsPoint,
12     QgsPointXY,
13     QgsProject,
14     QgsGeometry,
15     QgsMapRendererJob,
16     QgsWkbTypes,
17 )
18
19 from qgis.gui import (
20     QgsMapCanvas,
21     QgsVertexMarker,
22     QgsMapCanvasItem,
23     QgsMapMouseEvent,
24     QgsRubberBand,
25 )
```

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.

The map canvas is implemented with the `QgsMapCanvas` class in the `qgis.gui` module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom

graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please read the [overview of the framework](#).

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: defina `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 pre-definidas 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/data/data.gpkg|layername=airports", "Airports_
2 ↪layer", "ogr")
3 if not vlayer.isValid():
4     print("Layer failed to load!")
5
6 # add layer to the registry
7 QgsProject.instance().addMapLayer(vlayer)
8
9 # set extent to the extent of our layer
10 canvas.setExtent(vlayer.extent())
11
12 # set the map canvas layer set
13 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, QgsWkbTypes.LineGeometry) # line
points = [QgsPoint(-100, 45), QgsPoint(10, 60), QgsPoint(120, 45)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Para mostrar o polígono

```
r = QgsRubberBand(canvas, QgsWkbTypes.PolygonGeometry) # 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 panorâmicas 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()`.

```

1 from qgis.gui import *
2 from qgis.PyQt.QtWidgets import QAction, QMainWindow
3 from qgis.PyQt.QtCore import Qt
4
5 class MyWnd(QMainWindow):
6     def __init__(self, layer):
7         QMainWindow.__init__(self)
8
9         self.canvas = QgsMapCanvas()
10        self.canvas.setCanvasColor(Qt.white)
11
12        self.canvas.setExtent(layer.extent())
13        self.canvas.setLayers([layer])
14
15        self.setCentralWidget(self.canvas)
16
17        self.actionZoomIn = QAction("Zoom in", self)
18        self.actionZoomOut = QAction("Zoom out", self)
19        self.actionPan = QAction("Pan", self)
20
21        self.actionZoomIn.setCheckable(True)
22        self.actionZoomOut.setCheckable(True)
23        self.actionPan.setCheckable(True)
24
25        self.actionZoomIn.triggered.connect(self.zoomIn)
26        self.actionZoomOut.triggered.connect(self.zoomOut)
27        self.actionPan.triggered.connect(self.pan)
28
29        self.toolbar = self.addToolBar("Canvas actions")
30        self.toolbar.addAction(self.actionZoomIn)
31        self.toolbar.addAction(self.actionZoomOut)
32        self.toolbar.addAction(self.actionPan)
33
34        # create the map tools
35        self.toolPan = QgsMapToolPan(self.canvas)
36        self.toolPan.setAction(self.actionPan)
37        self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
38        self.toolZoomIn.setAction(self.actionZoomIn)
39        self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
40        self.toolZoomOut.setAction(self.actionZoomOut)
41
42        self.pan()
43
44        def zoomIn(self):
45            self.canvas.setMapTool(self.toolZoomIn)
46
47        def zoomOut(self):
48            self.canvas.setMapTool(self.toolZoomOut)
49
50        def pan(self):
51            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()
```

9.3.1 Select a feature using `QgsMapToolIdentifyFeature`

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

```
1 def callback(feature):
2     """Code called when the feature is selected by the user"""
3     print("You clicked on feature {}".format(feature.id()))
4
5 canvas = iface.mapCanvas()
6 feature_identifier = QgsMapToolIdentifyFeature(canvas)
7
8 # indicates the layer on which the selection will be done
9 feature_identifier.setLayer(vlayer)
10
11 # use the callback as a slot triggered when the user identifies a feature
12 feature_identifier.featureIdentified.connect(callback)
13
14 # activation of the map tool
15 canvas.setMapTool(feature_identifier)
```

9.3.2 Adicionar itens ao menu contextual da tela do mapa

A interação com a tela do mapa também pode ser feita através de entradas que você pode adicionar a seu menu contextual utilizando o menu `contextMenuAboutToShow` sinal.

O seguinte código acrescenta: 'menuselection:Meu menu -> Minha ação' ao lado das entradas padrão quando você clicar com o botão direito do mouse sobre a tela do mapa.

```
1 # a slot to populate the context menu
2 def populateContextMenu(menu: QMenu, event: QgsMapMouseEvent):
3     subMenu = menu.addMenu('My Menu')
4     action = subMenu.addAction('My Action')
5     action.triggered.connect(lambda *args:
6                             print(f'Action triggered at {event.x()}, {event.y()}'))
7
8 canvas.contextMenuAboutToShow.connect(populateContextMenu)
9 canvas.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.

```

1 class RectangleMapTool(QgsMapToolEmitPoint):
2     def __init__(self, canvas):
3         self.canvas = canvas
4         QgsMapToolEmitPoint.__init__(self, self.canvas)
5         self.rubberBand = QgsRubberBand(self.canvas, QgsWkbTypes.PolygonGeometry)
6         self.rubberBand.setColor(Qt.red)
7         self.rubberBand.setWidth(1)
8         self.reset()
9
10    def reset(self):
11        self.startPoint = self.endPoint = None
12        self.isEmittingPoint = False
13        self.rubberBand.reset(QgsWkbTypes.PolygonGeometry)
14
15    def canvasPressEvent(self, e):
16        self.startPoint = self.toMapCoordinates(e.pos())
17        self.endPoint = self.startPoint
18        self.isEmittingPoint = True
19        self.showRect(self.startPoint, self.endPoint)
20
21    def canvasReleaseEvent(self, e):
22        self.isEmittingPoint = False
23        r = self.rectangle()
24        if r is not None:
25            print("Rectangle:", r.xMinimum(),
26                  r.yMinimum(), r.xMaximum(), r.yMaximum()
27                  )
28
29    def canvasMoveEvent(self, e):
30        if not self.isEmittingPoint:
31            return
32
33        self.endPoint = self.toMapCoordinates(e.pos())
34        self.showRect(self.startPoint, self.endPoint)
35
36    def showRect(self, startPoint, endPoint):
37        self.rubberBand.reset(QgsWkbTypes.PolygonGeometry)
38        if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
39            return
40
41        point1 = QgsPointXY(startPoint.x(), startPoint.y())
42        point2 = QgsPointXY(startPoint.x(), endPoint.y())
43        point3 = QgsPointXY(endPoint.x(), endPoint.y())
44        point4 = QgsPointXY(endPoint.x(), startPoint.y())
45
46        self.rubberBand.addPoint(point1, False)
47        self.rubberBand.addPoint(point2, False)
48        self.rubberBand.addPoint(point3, False)
49        self.rubberBand.addPoint(point4, True)    # true to update canvas
50        self.rubberBand.show()
51
52    def rectangle(self):
53        if self.startPoint is None or self.endPoint is None:
54            return None
55        elif (self.startPoint.x() == self.endPoint.x() or \
56              self.startPoint.y() == self.endPoint.y()):
57            return None
58
59        return QgsRectangle(self.startPoint, self.endPoint)
60
61    def deactivate(self):
62        QgsMapTool.deactivate(self)

```

(continua na próxima página)

(continuação da página anterior)

```
63 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:

```
1 class CircleCanvasItem(QgsMapCanvasItem):
2     def __init__(self, canvas):
3         super().__init__(canvas)
4         self.center = QgsPoint(0, 0)
5         self.size = 100
6
7     def setCenter(self, center):
8         self.center = center
9
10    def center(self):
11        return self.center
12
13    def setSize(self, size):
14        self.size = size
15
16    def size(self):
17        return self.size
18
19    def boundingRect(self):
20        return QRectF(self.center.x() - self.size/2,
21                      self.center.y() - self.size/2,
22                      self.center.x() + self.size/2,
23                      self.center.y() + self.size/2)
24
25    def paint(self, painter, option, widget):
26        path = QPainterPath()
27        path.moveTo(self.center.x(), self.center.y());
28        path.arcTo(self.boundingRect(), 0.0, 360.0)
29        painter.fillPath(path, QColor("red"))
30
31
32    # Using the custom item:
33    item = CircleCanvasItem(iface.mapCanvas())
34    item.setCenter(QgsPointXY(200,200))
35    item.setSize(80)
```

Renderização em impressão de mapas

Dica: Os trechos de código nesta página precisam das seguintes importações:

```
1 import os
2
3 from qgis.core import (
4     QgsGeometry,
5     QgsMapSettings,
6     QgsPrintLayout,
7     QgsMapSettings,
8     QgsMapRendererParallelJob,
9     QgsLayoutItemLabel,
10    QgsLayoutItemLegend,
11    QgsLayoutItemMap,
12    QgsLayoutItemPolygon,
13    QgsLayoutItemScaleBar,
14    QgsLayoutExporter,
15    QgsLayoutItem,
16    QgsLayoutPoint,
17    QgsLayoutSize,
18    QgsUnitTypes,
19    QgsProject,
20    QgsFillSymbol,
21    QgsAbstractValidityCheck,
22    check,
23 )
24
25 from qgis.PyQt.QtGui import (
26     QPolygonF,
27     QColor,
28 )
29
30 from qgis.PyQt.QtCore import (
31     QPointF,
32     QRectF,
33     QSize,
34 )
```

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.

Aqui está um exemplo:

```
1 image_location = os.path.join(QgsProject.instance().homePath(), "render.png")
2
3 vlayer = iface.activeLayer()
4 settings = QgsMapSettings()
5 settings.setLayers([vlayer])
6 settings.setBackgroundColor(QColor(255, 255, 255))
7 settings.setOutputSize(QSize(800, 600))
8 settings.setExtent(vlayer.extent())
9
10 render = QgsMapRendererParallelJob(settings)
11
12 def finished():
13     img = render.renderedImage()
14     # save the image; e.g. img.save("/Users/myuser/render.png", "png")
15     img.save(image_location, "png")
16
17 render.finished.connect(finished)
18
19 # Start the rendering
20 render.start()
21
22 # The following loop is not normally required, we
23 # are using it here because this is a standalone example.
24 from qgis.PyQt.QtCore import QEventLoop
25 loop = QEventLoop()
26 render.finished.connect(loop.quit)
27 loop.exec_()
```

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

10.3 Saída usando layout de impressão

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

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

Isto inicializa o layout com algumas configurações padrão, especificamente adicionando uma página A4 vazia ao layout. Você pode criar layouts sem chamar o método `initializeDefaults()`, mas você mesmo terá que se encarregar de adicionar páginas ao layout.

The previous code creates a “temporary” layout that is not visible in the GUI. It can be handy to e.g. quickly add some items and export without modifying the project itself nor expose these changes to the user. If you want the layout to be saved/restored along with the project and available in the layout manager, then add:

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

Agora podemos adicionar vários elementos (mapa, rótulo, ...) 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.

- **mapa** — Aqui nós criamos um mapa de um tamanho personalizado e renderizamos a tela do mapa atual

```
1 map = QgsLayoutItemMap(layout)
2 # Set map item position and size (by default, it is a 0 width/0 height item,
  ↪ placed at 0,0)
3 map.attemptMove(QgsLayoutPoint(5,5, QgsUnitTypes.LayoutMillimeters))
4 map.attemptResize(QgsLayoutSize(200,200, QgsUnitTypes.LayoutMillimeters))
5 # Provide an extent to render
6 map.zoomToExtent(iface.mapCanvas().extent())
7 layout.addLayoutItem(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.addLayoutItem(label)
```

- **legenda**

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

- **barra de escala**

```

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

```

- forma baseada em nós

```

1 polygon = QPolygonF()
2 polygon.append(QPointF(0.0, 0.0))
3 polygon.append(QPointF(100.0, 0.0))
4 polygon.append(QPointF(200.0, 100.0))
5 polygon.append(QPointF(100.0, 200.0))
6
7 polygonItem = QgsLayoutItemPolygon(polygon, layout)
8 layout.addLayoutItem(polygonItem)
9
10 props = {}
11 props["color"] = "green"
12 props["style"] = "solid"
13 props["style_border"] = "solid"
14 props["color_border"] = "black"
15 props["width_border"] = "10.0"
16 props["joinstyle"] = "miter"
17
18 symbol = QgsFillSymbol.createSimple(props)
19 polygonItem.setSymbol(symbol)

```

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))

```

Uma moldura é desenhada ao redor de cada item por padrão. Você pode removê-la da seguinte forma:

```

# 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 Checking layout validity

A layout is a made of a set of interconnected items and it can happen that these connections are broken during modifications (a legend connected to a removed map, an image item with missing source file,...) or you may want to apply custom constraints to the layout items. The `QgsAbstractValidityCheck` helps you achieve this.

A basic check looks like:

```

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

```

Here's a check which throws a warning whenever a layout map item is set to the web mercator projection:

```

1 @check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
2 def layout_map_crs_choice_check(context, feedback):
3     layout = context.layout
4     results = []
5     for i in layout.items():
6         if isinstance(i, QgsLayoutItemMap) and i.crs().authid() == 'EPSG:3857':
7             res = QgsValidityCheckResult()
8             res.type = QgsValidityCheckResult.Warning
9             res.title = 'Map projection is misleading'
10            res.detailedDescription = 'The projection for the map item {} is set to <i>
↳ Web Mercator (EPSG:3857)</i> which misrepresents areas and shapes. Consider
↳ using an appropriate local projection instead.'.format(i.displayName())
11            results.append(res)
12
13     return results

```

And here's a more complex example, which throws a warning if any layout map items are set to a CRS which is only valid outside of the extent shown in that map item:

```

1 @check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
2 def layout_map_crs_area_check(context, feedback):
3     layout = context.layout
4     results = []
5     for i in layout.items():
6         if isinstance(i, QgsLayoutItemMap):
7             bounds = i.crs().bounds()
8             ct = QgsCoordinateTransform(QgsCoordinateReferenceSystem('EPSG:4326'),
↳ i.crs(), QgsProject.instance())
9             bounds_crs = ct.transformBoundingBox(bounds)
10
11            if not bounds_crs.contains(i.extent()):
12                res = QgsValidityCheckResult()
13                res.type = QgsValidityCheckResult.Warning
14                res.title = 'Map projection is incorrect'
15                res.detailedDescription = 'The projection for the map item {} is
↳ set to \'{}\ ', which is not valid for the area displayed within the map.'.
↳ format(i.displayName(), i.crs().authid())
16                results.append(res)
17
18     return results

```

10.3.2 Exportando o layout

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

```

1 base_path = os.path.join(QgsProject.instance().homePath())
2 pdf_path = os.path.join(base_path, "output.pdf")
3
4 exporter = QgsLayoutExporter(layout)
5 exporter.exportToPdf(pdf_path, QgsLayoutExporter.PdfExportSettings())

```

Use `exportToSvg()` or `exportToImage()` in case you want to export to respectively an SVG or image file instead of a PDF file.

10.3.3 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.

Expressões, filtragem e cálculo dos valores

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

```
1 from qgis.core import (  
2     edit,  
3     QgsExpression,  
4     QgsExpressionContext,  
5     QgsFeature,  
6     QgsFeatureRequest,  
7     QgsField,  
8     QgsFields,  
9     QgsVectorLayer,  
10    QgsPointXY,  
11    QgsGeometry,  
12    QgsProject,  
13    QgsExpressionContextUtils  
14 )
```

O QGIS tem suporte para analisar expressões do tipo SQL. Apenas um pequeno subconjunto da sintaxe SQL é suportada. As expressões podem ser avaliadas como predicados booleanos (retornando `True` ou `False`) 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: `'hello world'`
- coluna de 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`
- `sin(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 Analisando Expressões

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

```
1 exp = QgsExpression('1 + 1 = 2')
2 assert(not exp.hasParserError())
3
4 exp = QgsExpression('1 + 1 = ')
5 assert(exp.hasParserError())
6
7 assert(exp.parserErrorMessage() == '\nsyntax error, unexpected end of file')
```

11.2 Avaliando expressões

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

Esta expressão básica avalia uma simples operação aritmética:

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

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

A expressão também pode ser usada para comparação, avaliando com 1 (True) ou 0 (False)

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

11.2.2 Expressões com feições

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

O exemplo a seguir mostra como criar uma feição com um campo chamado “Column” e como adicionar essa feição ao contexto da expressão.

```
1 fields = QgsFields()
2 field = QgsField('Column')
3 fields.append(field)
4 feature = QgsFeature()
5 feature.setFields(fields)
6 feature.setAttribute(0, 99)
7
8 exp = QgsExpression('"Column"')
9 context = QgsExpressionContext()
10 context.setFeature(feature)
11 exp.evaluate(context)
12 # 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:

```
1 from qgis.PyQt.QtCore import QMetaType
2
3 # create a vector layer
4 vl = QgsVectorLayer("Point", "Companies", "memory")
5 pr = vl.dataProvider()
6 pr.addAttributes([QgsField("Name", QMetaType.Type.QString),
7                   QgsField("Employees", QMetaType.Type.Int),
8                   QgsField("Revenue", QMetaType.Type.Double),
9                   QgsField("Rev. per employee", QMetaType.Type.Double),
10                  QgsField("Sum", QMetaType.Type.Double),
11                  QgsField("Fun", QMetaType.Type.Double)])
12 vl.updateFields()
13
14 # add data to the first three fields
15 my_data = [
16     {'x': 0, 'y': 0, 'name': 'ABC', 'emp': 10, 'rev': 100.1},
17     {'x': 1, 'y': 1, 'name': 'DEF', 'emp': 2, 'rev': 50.5},
18     {'x': 5, 'y': 5, 'name': 'GHI', 'emp': 100, 'rev': 725.9}]
19
```

(continua na próxima página)

(continuação da página anterior)

```

20 for rec in my_data:
21     f = QgsFeature()
22     pt = QgsPointXY(rec['x'], rec['y'])
23     f.setGeometry(QgsGeometry.fromPointXY(pt))
24     f.setAttributes([rec['name'], rec['emp'], rec['rev']])
25     pr.addFeature(f)
26
27 vl.updateExtents()
28 QgsProject.instance().addMapLayer(vl)
29
30 # The first expression computes the revenue per employee.
31 # The second one computes the sum of all revenue values in the layer.
32 # The final third expression doesn't really make sense but illustrates
33 # the fact that we can use a wide range of expression functions, such
34 # as area and buffer in our expressions:
35 expression1 = QgsExpression('"Revenue"/"Employees"')
36 expression2 = QgsExpression('sum("Revenue")')
37 expression3 = QgsExpression('area(buffer($geometry, "Employees"))')
38
39 # QgsExpressionContextUtils.globalProjectLayerScopes() is a convenience
40 # function that adds the global, project, and layer scopes all at once.
41 # Alternatively, those scopes can also be added manually. In any case,
42 # it is important to always go from "most generic" to "most specific"
43 # scope, i.e. from global to project to layer
44 context = QgsExpressionContext()
45 context.appendScopes(QgsExpressionContextUtils.globalProjectLayerScopes(vl))
46
47 with edit(vl):
48     for f in vl.getFeatures():
49         context.setFeature(f)
50         f['Rev. per employee'] = expression1.evaluate(context)
51         f['Sum'] = expression2.evaluate(context)
52         f['Fun'] = expression3.evaluate(context)
53         vl.updateFeature(f)
54
55 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 feição que corresponda a um predicado.

```

1 layer = QgsVectorLayer("Point?field=Test:integer",
2                         "addfeat", "memory")
3
4 layer.startEditing()
5
6 for i in range(10):
7     feature = QgsFeature()
8     feature.setAttributes([i])
9     assert(layer.addFeature(feature))
10 layer.commitChanges()
11
12 expression = 'Test >= 3'
13 request = QgsFeatureRequest().setFilterExpression(expression)
14
15 matches = 0

```

(continua na próxima página)

(continuação da página anterior)

```
16 for f in layer.getFeatures(request):  
17     matches += 1  
18  
19 print(matches)
```

7

11.3 Manipulando erros de expressão

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

```
1 exp = QgsExpression("1 + 1 = 2")  
2 if exp.hasParserError():  
3     raise Exception(exp.parserErrorString())  
4  
5 value = exp.evaluate()  
6 if exp.hasEvalError():  
7     raise ValueError(exp.evalErrorString())
```


Leitura e Armazenamento de Configurações

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

```
1 from qgis.core import (  
2     QgsProject,  
3     QgsSettings,  
4     QgsVectorLayer  
5 )
```

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 de interesse. 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.

```
1 def store():  
2     s = QgsSettings()  
3     s.setValue("myplugin/mytext", "hello world")  
4     s.setValue("myplugin/myint", 10)  
5     s.setValue("myplugin/myreal", 3.14)  
6  
7 def read():  
8     s = QgsSettings()  
9     mytext = s.value("myplugin/mytext", "default text")  
10    myint = s.value("myplugin/myint", 123)  
11    myreal = s.value("myplugin/myreal", 2.71)  
12    nonexistent = s.value("myplugin/nonexistent", None)
```

(continua na próxima página)

(continuação da página anterior)

```

13 print(mytext)
14 print(myint)
15 print(myreal)
16 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.

For a method to pre-configure the default values of the global settings through the `qgis_global_settings.ini` file, see [deploying_organization](#) for further details.

- **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.

```

1 proj = QgsProject.instance()
2
3 # store values
4 proj.writeEntry("myplugin", "mytext", "hello world")
5 proj.writeEntry("myplugin", "myint", 10)
6 proj.writeEntryDouble("myplugin", "mydouble", 0.01)
7 proj.writeEntryBool("myplugin", "mybool", True)
8
9 # read values (returns a tuple with the value, and a status boolean
10 # which communicates whether the value retrieved could be converted to
11 # its type, in these cases a string, an integer, a double and a boolean
12 # respectively)
13
14 mytext, type_conversion_ok = proj.readEntry("myplugin",
15                                             "mytext",
16                                             "default text")
17 myint, type_conversion_ok = proj.readNumEntry("myplugin",
18                                              "myint",
19                                              123)
20 mydouble, type_conversion_ok = proj.readDoubleEntry("myplugin",
21                                                     "mydouble",
22                                                     123)
23 mybool, type_conversion_ok = proj.readBoolEntry("myplugin",
24                                                  "mybool",
25                                                  123)

```

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

- **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()`.

```

1 vlayer = QgsVectorLayer()
2 # save a value
3 vlayer.setCustomProperty("mytext", "hello world")
4
5 # read the value again (returning "default text" if not found)
6 mytext = vlayer.customProperty("mytext", "default text")

```


Comunicação com o usuário

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

```
1 from qgis.core import (
2     QgsMessageLog,
3     QgsGeometry,
4 )
5
6 from qgis.gui import (
7     QgsMessageBar,
8 )
9
10 from qgis.PyQt.QtWidgets import (
11     QSizePolicy,
12     QPushButton,
13     QDialog,
14     QGridLayout,
15     QDialogButtonBox,
16 )
```

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 Qgs
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
```

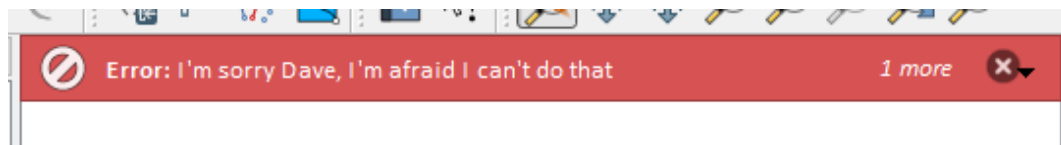


Figura13.1: Barra de mensagem do QGIS

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

```
iface.messageBar().pushMessage("Oops", "The plugin is not working as it should",  
level=Qgis.Critical, duration=3)
```

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

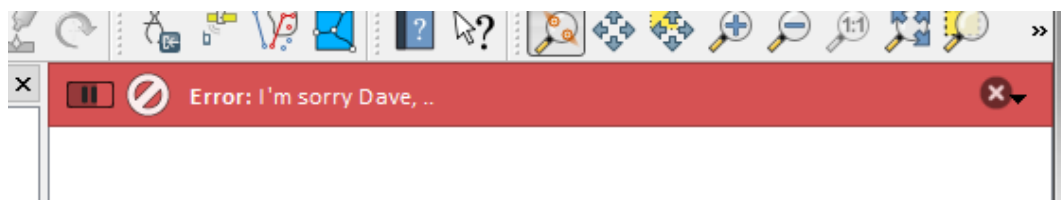


Figura13.2: Barra de Mensagem QGIS como temporizar

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

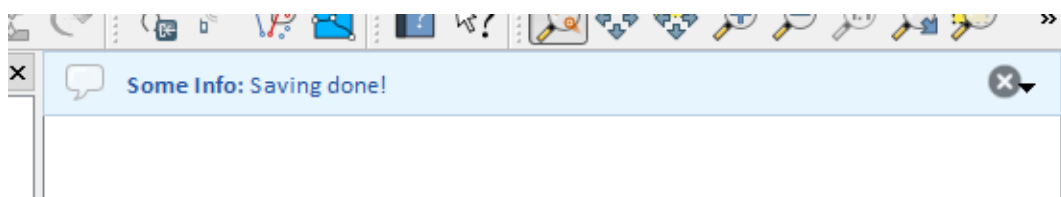


Figura13.3: Barra de mensagens QGIS (informação)

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

```
1 def showError():
2     pass
3
4 widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
5 button = QPushButton(widget)
6 button.setText("Show Me")
7 button.pressed.connect(showError)
8 widget.layout().addWidget(button)
9 iface.messageBar().pushWidget(widget, Qgis.Warning)
```

Messages(1): Missing Layers : Show Me

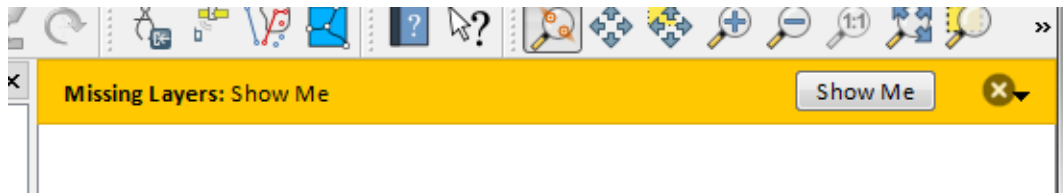


Figura13.4: Barra de mensagens QGIS com um botão

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

```

1 class MyDialog(QDialog):
2     def __init__(self):
3         QDialog.__init__(self)
4         self.bar = QgsMessageBar()
5         self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
6         self.setLayout(QGridLayout())
7         self.layout().setContentsMargins(0, 0, 0, 0)
8         self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
9         self.buttonbox.accepted.connect(self.run)
10        self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
11        self.layout().addWidget(self.bar, 0, 0, 1, 1)
12    def run(self):
13        self.bar.pushMessage("Hello", "World", level=Qgis.Info)
14
15 myDlg = MyDialog()
16 myDlg.show()

```

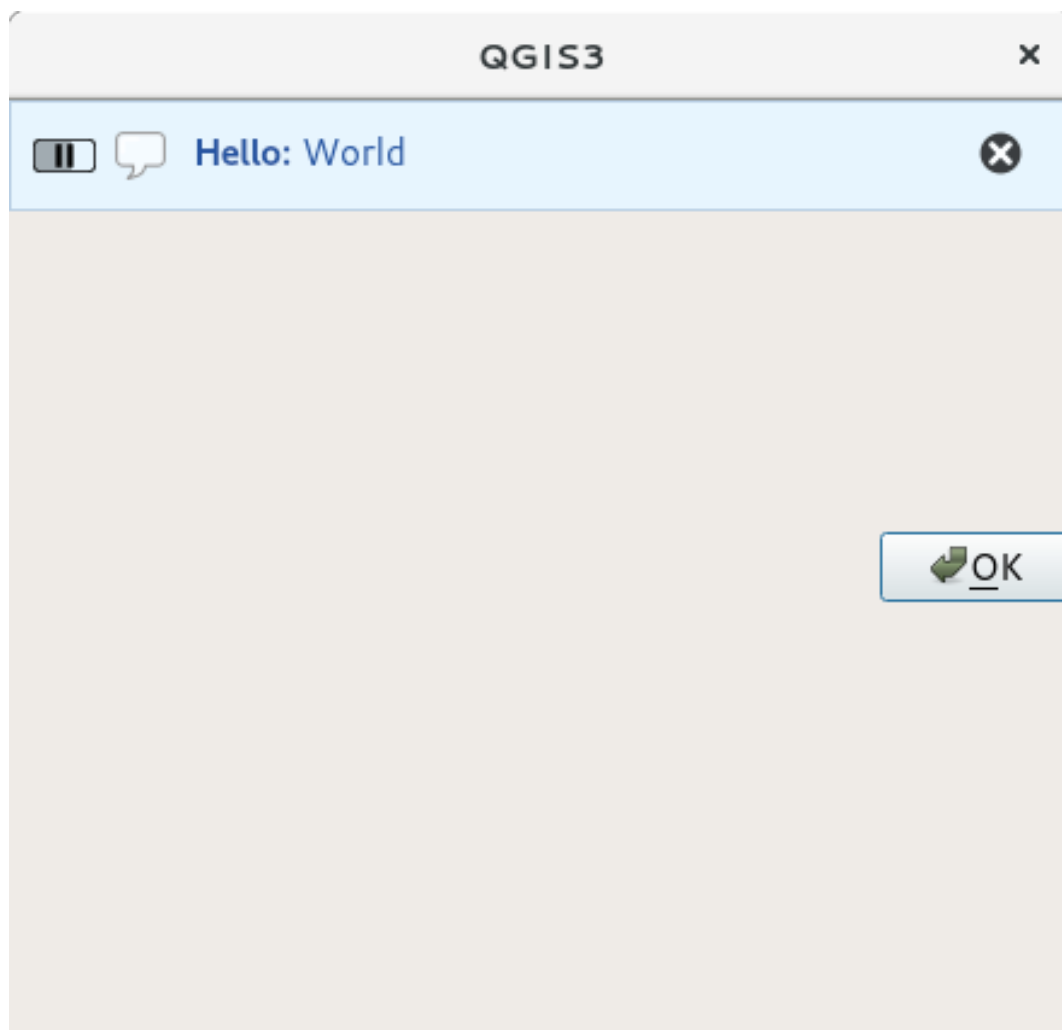


Figura13.5: Barra de mensagens QGIS em diálogo personalizado

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.

```
1 import time
2 from qgis.PyQt.QtWidgets import QProgressBar
3 from qgis.PyQt.QtCore import *
4 progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
5 progress = QProgressBar()
6 progress.setMaximum(10)
7 progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
8 progressMessageBar.layout().addWidget(progress)
9 iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)
10
11 for i in range(10):
12     time.sleep(1)
13     progress.setValue(i + 1)
14
15 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:

```
1 vlayer = iface.activeLayer()
2
3 count = vlayer.featureCount()
4 features = vlayer.getFeatures()
5
6 for i, feature in enumerate(features):
7     # do something time-consuming here
8     print('.') # printing should give enough time to present the progress
9
10    percent = i / float(count) * 100
11    # iface.mainWindow().statusBar().showMessage("Processed {} %".
    ↳format(int(percent)))
12    iface.statusBarIface().showMessage("Processed {} %".format(int(percent)))
13
14 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.

Exemplos para os diferentes tipos de extração são mostrados nas seguintes seções abaixo.

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 O python construído no módulo de extração

```
1 import logging
2 formatter = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
3 logfilename=r'c:\temp\example.log'
4 logging.basicConfig(filename=logfilename, level=logging.DEBUG, format=formatter)
5 logging.info("This logging info text goes into the file")
6 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
```

Se você quiser apagar o arquivo de log toda vez que executar seu script, você pode fazer algo como:

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

Outros recursos sobre como utilizar a instalação de extração em python estão disponíveis em:

- <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.

Infraestrutura de autenticação

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

```
1 from qgis.core import (
2     QgsApplication,
3     QgsRasterLayer,
4     QgsAuthMethodConfig,
5     QgsDataSourceUri,
6     QgsPkiBundle,
7     QgsMessageLog,
8 )
9
10 from qgis.gui import (
11     QgsAuthAuthoritiesEditor,
12     QgsAuthConfigEditor,
13     QgsAuthConfigSelect,
14     QgsAuthSettingsWidget,
15 )
16
17 from qgis.PyQt.QtWidgets import (
18     QWidget,
19     QTabWidget,
20 )
21
22 from qgis.PyQt.QtNetwork import QSslCertificate
```

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: Devido às restrições de segurança que foram levadas em consideração durante o projeto da infraestrutura de autenticação, apenas um subconjunto selecionado dos métodos internos é exposto ao Python.

14.2 Glossário

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

Master Password

Senha para permitir acesso e descryptografia 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.

Configuração de 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.

```

1 authMgr = QgsApplication.authManager()
2
3 # check if QgsAuthManager has already been initialized... a side effect
4 # of the QgsAuthManager.init() is that AuthDbPath is set.
5 # QgsAuthManager.init() is executed during QGIS application init and hence
6 # you do not normally need to call it directly.
7 if authMgr.authenticationDatabasePath():
8     # already initialized => we are inside a QGIS app.
9     if authMgr.masterPasswordIsSet():
10         msg = 'Authentication master password not recognized'
11         assert authMgr.masterPasswordSame("your master password"), msg
12     else:
13         msg = 'Master password could not be set'
14         # The verify parameter checks if the hash of the password was
15         # already saved in the authentication db
16         assert authMgr.setMasterPassword("your master password",
17                                         verify=True), msg
18 else:
19     # outside qgis, e.g. in a testing environment => setup env var before
20     # db init
21     os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
22     msg = 'Master password could not be set'
23     assert authMgr.setMasterPassword("your master password", True), msg
24     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` is the base class for any *Authentication Method*. Any Authentication Method sets a configuration hash map where authentication information will be stored. Hereafter a useful snippet to store PKI-path credentials for a hypothetical alice user:

```

1 authMgr = QgsApplication.authManager()
2 # set alice PKI data
3 config = QgsAuthMethodConfig()
4 config.setName("alice")

```

(continua na próxima página)

(continuação da página anterior)

```

5 config.setMethod("PKI-Paths")
6 config.setUri("https://example.com")
7 config.setConfig("certpath", "path/to/alice-cert.pem" )
8 config.setConfig("keypath", "path/to/alice-key.pem" )
9 # check if method parameters are correctly set
10 assert config.isValid()
11
12 # register alice data in authdb returning the ``authcfg`` of the stored
13 # configuration
14 authMgr.storeAuthenticationConfig(config)
15 newAuthCfgId = config.id()
16 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. EsriToken ESRI token based authentication
3. Identity-Cert Autenticação de certificado de identidade
4. OAuth2 OAuth2 authentication
5. PKI-Paths autenticação de caminhos PKI
6. PKI-PKCS#12 PKI PKCS#12 autenticação

Preencher Autoridades

```

1 authMgr = QgsApplication.authManager()
2 # add authorities
3 cacerts = QSslCertificate.fromPath( "/path/to/ca_chains.pem" )
4 assert cacerts is not None
5 # store CA
6 authMgr.storeCertAuthorities(cacerts)
7 # and rebuild CA caches
8 authMgr.rebuildCaCertsCache()
9 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:

```

1 # add alice cert in case of key with pwd
2 caBundlesList = [] # List of CA bundles
3 bundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
4                                     "/path/to/alice-key_w-pass.pem",
5                                     "unlock_pwd",
6                                     caBundlesList )
7 assert bundle is not None
8 # You can check bundle validity by calling:
9 # 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 Configs* 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: Take into account that not all QGIS data providers are integrated with the Authentication infrastructure. Each authentication method, derived from the base class `QgsAuthMethod` and support a different set of Providers. For example the `certIdentity()` method supports the following list of providers:

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

Sample output:

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

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:

```
1 authCfg = 'fm1s770'
2 quri = QgsDataSourceUri()
3 quri.setParam("layers", 'usa:states')
4 quri.setParam("styles", '')
5 quri.setParam("format", 'image/png')
6 quri.setParam("crs", 'EPSG:4326')
7 quri.setParam("dpiMode", '7')
8 quri.setParam("featureCount", '10')
9 quri.setParam("authcfg", authCfg) # <---- here my authCfg url parameter
10 quri.setParam("contextualWMSLegend", '0')
11 quri.setParam("url", 'https://my_auth_enabled_server_ip/wms')
12 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 `enablelib2` 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:

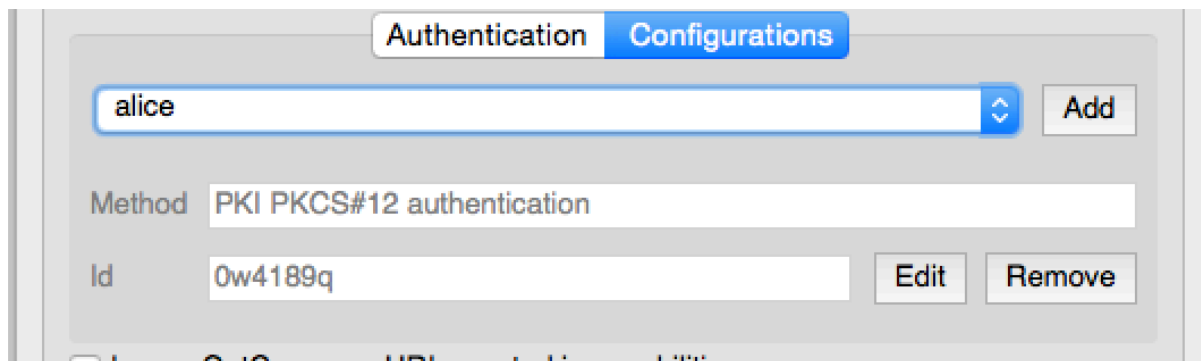
```
1 http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
  ↳FailedRequestError)
2 try:
3     response, content = http.request( "my_rest_url" )
4 except My_FailedRequestError, e:
5     # Handle exception
6     pass
```

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:

```

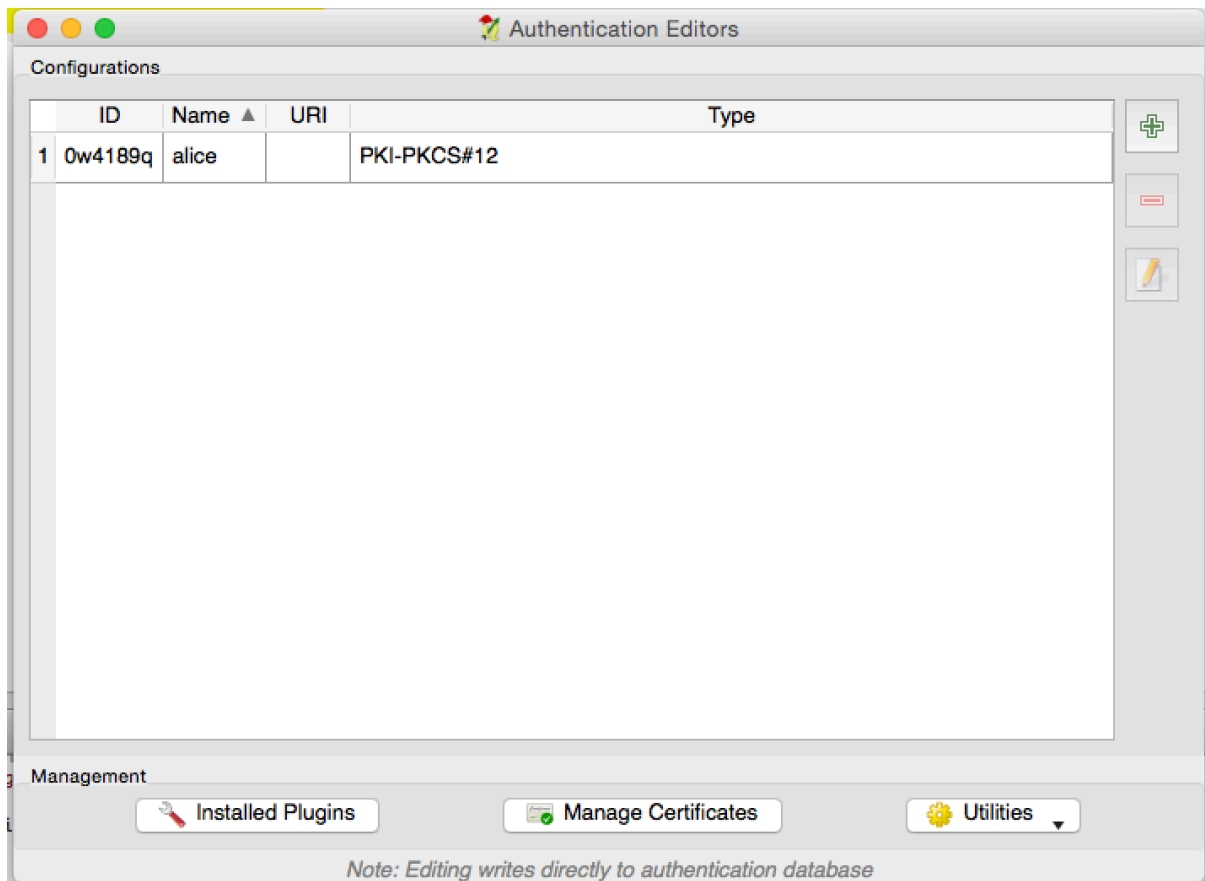
1 # create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
2 # the widget referred with `parent`
3 parent = QWidget() # Your GUI parent widget
4 gui = QgsAuthConfigSelect( parent, "postgres" )
5 # add the above created gui in a new tab of the interface where the
6 # GUI has to be integrated
7 tabGui = QTabWidget()
8 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 Methods* 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:

```

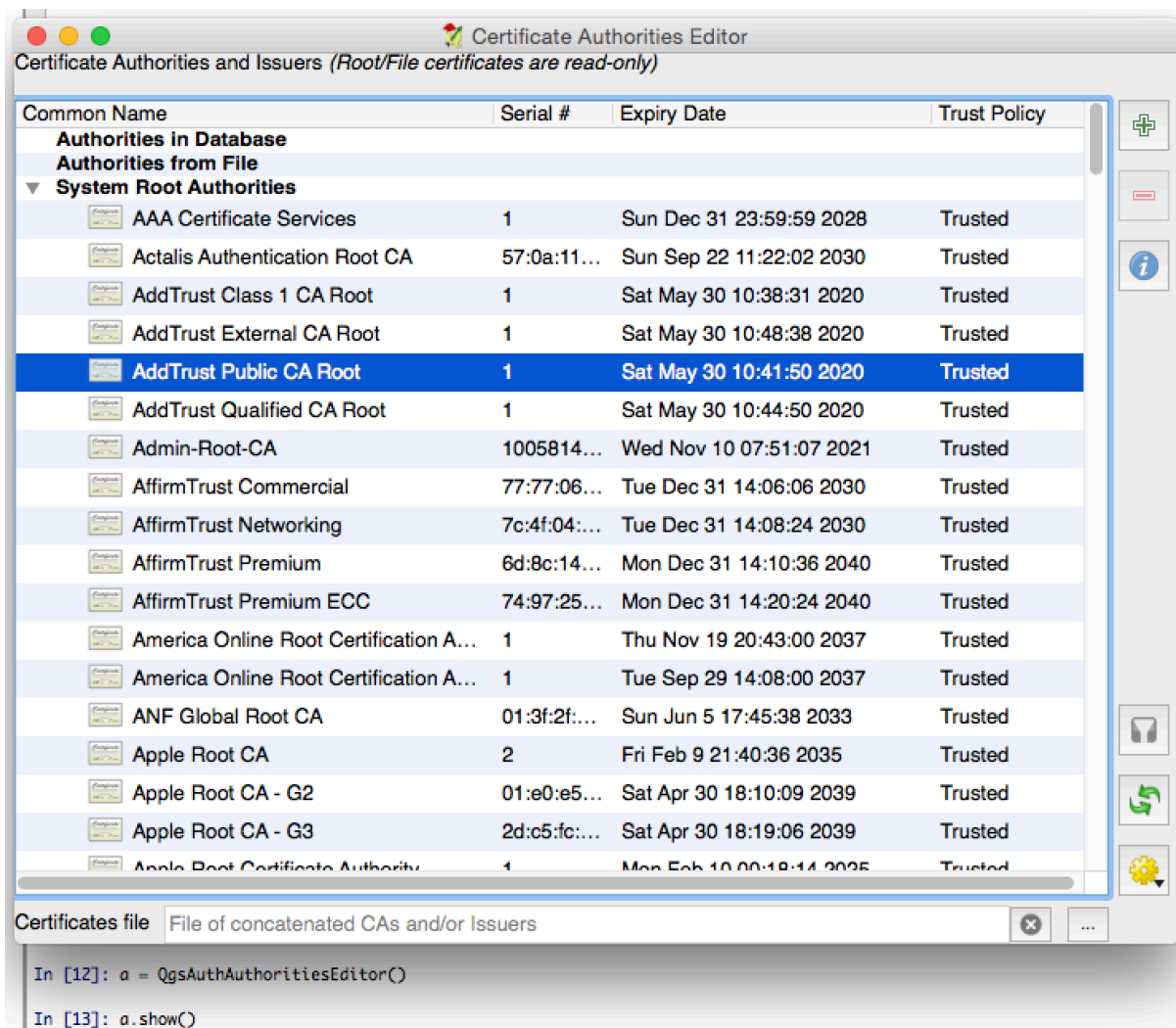
1 # create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
2 # the widget referred with `parent`
3 parent = QWidget() # Your GUI parent widget
4 gui = QgsAuthConfigSelect( parent )
5 gui.show()

```

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:

```
1 # create the instance of the QgsAuthAuthoritiesEditor GUI hierarchically
2 # linked to the widget referred with 'parent'
3 parent = QWidget() # Your GUI parent widget
4 gui = QgsAuthAuthoritiesEditor( parent )
5 gui.show()
```

Tarefas - trabalho pesado em segundo plano

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

```
1 from qgis.core import (
2     Qgis,
3     QgsApplication,
4     QgsMessageLog,
5     QgsProcessingAlgRunnerTask,
6     QgsProcessingContext,
7     QgsProcessingFeedback,
8     QgsProject,
9     QgsTask,
10    QgsTaskManager,
11 )
```

15.1 Introdução

O processamento em segundo plano usando threads é uma maneira de manter uma interface de usuário respondendo 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

```
1 def heavyFunction():
2     # Some CPU intensive processing ...
3     pass
4
5 def workdone():
6     # ... do something useful with the results
7     pass
8
9 task = QgsTask.fromFunction('heavy function', heavyFunction,
10                             on_finished=workdone)
```

- Criar uma tarefa a partir de um algoritmo de processamento

```
1 params = dict()
2 context = QgsProcessingContext()
3 context.setProject(QgsProject.instance())
4 feedback = QgsProcessingFeedback()
5
6 buffer_alg = QgsApplication.instance().processingRegistry().algorithmById(
7     ↪ 'native:buffer')
8 task = QgsProcessingAlgRunnerTask(buffer_alg, params, context,
9                                   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.

Moreover always make sure that `context` and `feedback` live for at least as long as the tasks that use them. QGIS will crash if, upon completion of a task, `QgsTaskManager` fails to access the `context` and `feedback` against which the task was scheduled.

Nota: It is a common pattern to call `setProject()` shortly after calling `QgsProcessingContext`. This allows the task as well as its callback function to use most of the project-wide settings. This is especially valuable when working with spatial layers in the callback function.

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

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

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

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, Qgis
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
19
20     def run(self):
21         """Here you implement your heavy lifting.
22         Should periodically test for isCanceled() to gracefully
23         abort.
24         This method MUST return True or False.
25         Raising exceptions will crash QGIS, so we handle them
26         internally and raise them in self.finished
27         """
28         QgsMessageLog.logMessage('Started task "{}".format(
29             self.description(),
30             MESSAGE_CATEGORY, Qgis.Info)
31         wait_time = self.duration / 100
32         for i in range(100):
33             sleep(wait_time)
34             # use setProgress to report progress
35             self.setProgress(i)
36             arandominteger = random.randint(0, 500)
37             self.total += arandominteger
38             self.iterations += 1
39             # check isCanceled() to handle cancellation
40             if self.isCanceled():
41                 return False
42             # simulate exceptions to show how to abort task
43             if arandominteger == 42:
44                 # DO NOT raise Exception('bad value!')
45                 # this would crash QGIS
46                 self.exception = Exception('bad value!')
47                 return False
48             return True
49
50     def finished(self, result):
51         """
52         This function is automatically called when the task has

```

(continua na próxima página)

```

53         completed (successfully or not).
54         You implement finished() to do whatever follow-up stuff
55         should happen after the task is complete.
56         finished is always called from the main thread, so it's safe
57         to do GUI operations and raise Python exceptions here.
58         result is the return value from self.run.
59         """
60         if result:
61             QgsMessageLog.logMessage(
62                 'RandomTask "{name}" completed\n' \
63                 'RandomTotal: {total} (with {iterations} '\
64                 'iterations)'.format(
65                     name=self.description(),
66                     total=self.total,
67                     iterations=self.iterations),
68                 MESSAGE_CATEGORY, Qgs.Success)
69         else:
70             if self.exception is None:
71                 QgsMessageLog.logMessage(
72                     'RandomTask "{name}" not successful but without '\
73                     'exception (probably the task was manually '\
74                     'canceled by the user)'.format(
75                         name=self.description(),
76                         MESSAGE_CATEGORY, Qgs.Warning)
77             else:
78                 QgsMessageLog.logMessage(
79                     'RandomTask "{name}" Exception: {exception}'.format(
80                         name=self.description(),
81                         exception=self.exception),
82                     MESSAGE_CATEGORY, Qgs.Critical)
83             raise self.exception
84
85         def cancel(self):
86             QgsMessageLog.logMessage(
87                 'RandomTask "{name}" was canceled'.format(
88                     name=self.description()),
89                     MESSAGE_CATEGORY, Qgs.Info)
90             super().cancel()
91
92
93 longtask = RandomIntegerSumTask('waste cpu long', 20)
94 shorttask = RandomIntegerSumTask('waste cpu short', 10)
95 minitask = RandomIntegerSumTask('waste cpu mini', 5)
96 shortsubtask = RandomIntegerSumTask('waste cpu subtask short', 5)
97 longsubtask = RandomIntegerSumTask('waste cpu subtask long', 10)
98 shortestsubtask = RandomIntegerSumTask('waste cpu subtask shortest', 4)
99
100 # Add a subtask (shortsubtask) to shorttask that must run after
101 # minitask and longtask has finished
102 shorttask.addSubTask(shortsubtask, [minitask, longtask])
103 # Add a subtask (longsubtask) to longtask that must be run
104 # before the parent task
105 longtask.addSubTask(longsubtask, [], QgsTask.ParentDependsOnSubTask)
106 # Add a subtask (shortestsubtask) to longtask
107 longtask.addSubTask(shortestsubtask)
108
109 QgsApplication.taskManager().addTask(longtask)
110 QgsApplication.taskManager().addTask(shorttask)
111 QgsApplication.taskManager().addTask(minitask)

```

```

1 RandomIntegerSumTask(0): Started task "waste cpu subtask shortest"
2 RandomIntegerSumTask(0): Started task "waste cpu short"
3 RandomIntegerSumTask(0): Started task "waste cpu mini"
4 RandomIntegerSumTask(0): Started task "waste cpu subtask long"
5 RandomIntegerSumTask(3): Task "waste cpu subtask shortest" completed
6 RandomTotal: 25452 (with 100 iterations)
7 RandomIntegerSumTask(3): Task "waste cpu mini" completed
8 RandomTotal: 23810 (with 100 iterations)
9 RandomIntegerSumTask(3): Task "waste cpu subtask long" completed
10 RandomTotal: 26308 (with 100 iterations)
11 RandomIntegerSumTask(0): Started task "waste cpu long"
12 RandomIntegerSumTask(3): Task "waste cpu long" completed
13 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`.

```

1 import random
2 from time import sleep
3
4 MESSAGE_CATEGORY = 'TaskFromFunction'
5
6 def doSomething(task, wait_time):
7     """
8     Raises an exception to abort the task.
9     Returns a result if success.
10    The result will be passed, together with the exception (None in
11    the case of success), to the on_finished method.
12    If there is an exception, there will be no result.
13    """
14    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
15                             MESSAGE_CATEGORY, QgsInfo.Info)
16    wait_time = wait_time / 100
17    total = 0
18    iterations = 0
19    for i in range(100):
20        sleep(wait_time)
21        # use task.setProgress to report progress
22        task.setProgress(i)
23        arandominteger = random.randint(0, 500)
24        total += arandominteger
25        iterations += 1
26        # check task.isCanceled() to handle cancellation
27        if task.isCanceled():
28            stopped(task)
29            return None
30        # raise an exception to abort the task
31        if arandominteger == 42:
32            raise Exception('bad value!')
33    return {'total': total, 'iterations': iterations,
34           'task': task.description()}
35
36 def stopped(task):
37    QgsMessageLog.logMessage(
38        'Task "{name}" was canceled'.format(
39            name=task.description()),

```

(continua na próxima página)

```

40     MESSAGE_CATEGORY, Qgis.Info)
41
42 def completed(exception, result=None):
43     """This is called when doSomething is finished.
44     Exception is not None if doSomething raises an exception.
45     result is the return value of doSomething."""
46     if exception is None:
47         if result is None:
48             QgsMessageLog.logMessage(
49                 'Completed with no exception and no result '\
50                 '(probably manually canceled by the user)',
51                 MESSAGE_CATEGORY, Qgis.Warning)
52         else:
53             QgsMessageLog.logMessage(
54                 'Task {name} completed\n'
55                 'Total: {total} ( with {iterations} '
56                 'iterations)'.format(
57                     name=result['task'],
58                     total=result['total'],
59                     iterations=result['iterations']),
60                 MESSAGE_CATEGORY, Qgis.Info)
61     else:
62         QgsMessageLog.logMessage("Exception: {}".format(exception),
63                                 MESSAGE_CATEGORY, Qgis.Critical)
64     raise exception
65
66 # Create a few tasks
67 task1 = QgsTask.fromFunction('Waste cpu 1', doSomething,
68                             on_finished=completed, wait_time=4)
69 task2 = QgsTask.fromFunction('Waste cpu 2', doSomething,
70                             on_finished=completed, wait_time=3)
71 QgsApplication.taskManager().addTask(task1)
72 QgsApplication.taskManager().addTask(task2)

```

```

1 RandomIntegerSumTask(0): Started task "waste cpu subtask short"
2 RandomTaskFromFunction(0): Started task Waste cpu 1
3 RandomTaskFromFunction(0): Started task Waste cpu 2
4 RandomTaskFromFunction(0): Task Waste cpu 2 completed
5 RandomTotal: 23263 ( with 100 iterations)
6 RandomTaskFromFunction(0): Task Waste cpu 1 completed
7 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.

```

1 from functools import partial
2 from qgis.core import (QgsTaskManager, QgsMessageLog,
3                       QgsProcessingAlgRunnerTask, QgsApplication,
4                       QgsProcessingContext, QgsProcessingFeedback,
5                       QgsProject)
6
7 MESSAGE_CATEGORY = 'AlgRunnerTask'
8
9 def task_finished(context, successful, results):
10     if not successful:
11         QgsMessageLog.logMessage('Task finished unsuccessfully',
12                                 MESSAGE_CATEGORY, Qgis.Warning)

```

(continua na próxima página)

(continuação da página anterior)

```

13     output_layer = context.getMapLayer(results['OUTPUT'])
14     # because getMapLayer doesn't transfer ownership, the layer will
15     # be deleted when context goes out of scope and you'll get a
16     # crash.
17     # takeMapLayer transfers ownership so it's then safe to add it
18     # to the project and give the project ownership.
19     if output_layer and output_layer.isValid():
20         QgsProject.instance().addMapLayer(
21             context.takeResultLayer(output_layer.id()))
22
23 alg = QgsApplication.processingRegistry().algorithmById(
24     'qgis:randompointsinextent')
25
26 # `context` and `feedback` need to
27 # live for as least as long as `task`,
28 # otherwise the program will crash.
29 # Initializing them globally is a sure way
30 # of avoiding this unfortunate situation.
31 context = QgsProcessingContext()
32 feedback = QgsProcessingFeedback()
33 params = {
34     'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
35     'MIN_DISTANCE': 0.0,
36     'POINTS_NUMBER': 50000,
37     'TARGET_CRS': 'EPSG:4326',
38     'OUTPUT': 'memory:My random points'
39 }
40 task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
41 task.executed.connect(partial(task_finished, context))
42 QgsApplication.taskManager().addTask(task)

```

See also: <https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/>.

Desenvolvendo complementos Python

16.1 Estruturando Complementos Python

The main steps for creating a plugin are:

1. *Idea*: Have an idea about what you want to do with your new QGIS plugin.
2. *Setup*: *Create the files for your plugin*. Depending on the plugin type, some are mandatory while others are optional
3. *Develop*: *Write the code* in appropriate files
4. *Document*: *Write the plugin documentation*
5. Optionally: *Translate*: *Translate your plugin* into different languages
6. *Teste*: *Reload your plugin* para verificar se tudo está OK
7. *Publicando*: publique seu complemento no repositório QGIS ou faça seu próprio repositório como um “arsenal” de “armas GIS” pessoais.

16.1.1 Getting started

Before starting to write a new plugin, have a look at the *Repositório Oficial de complementos Python*. The source code of existing plugins can help you to learn more about programming. You may also find that a similar plugin already exists and you may be able to extend it or at least build on it to develop your own.

Set up plugin file structure

To get started with a new plugin, we need to set up the necessary plugin files.

There are two plugin template resources that can help get you started:

- For educational purposes or whenever a minimalist approach is desired, the [minimal plugin template](#) provides the basic files (skeleton) necessary to create a valid QGIS Python plugin.
- For a more fully feature plugin template, the [Plugin Builder](#) can create templates for multiple different plugin types, including features such as localization (translation) and testing.

A typical plugin directory includes the following files:

- `metadata.txt` - *required* - Contains general info, version, name and some other metadata used by plugins website and plugin infrastructure.
- `__init__.py` - *required* - The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.py` - *core code* - The main working code of the plugin. Contains all the information about the actions of the plugin and the main code.
- `form.ui` - *for plugins with custom GUI* - The GUI created by Qt Designer.
- `form.py` - *compiled GUI* - The translation of the `form.ui` described above to Python.
- `resources.qrc` - *optional* - An .xml document created by Qt Designer. Contains relative paths to resources used in the GUI forms.
- `resources.py` - *compiled resources, optional* - The translation of the .qrc file described above to Python.
- `LICENSE` - *required* if plugin is to be published or updated in the QGIS Plugins Directory, otherwise *optional*. File should be a plain text file with no file extension in the filename.

Aviso: If you plan to upload the plugin to the [Repositório Oficial de complementos Python](#) you must check that your plugin follows some additional rules, required for plugin [Validação](#).

16.1.2 Writing plugin code

The following section shows what content should be added in each of the files introduced above.

metadata.txt

First, the Plugin Manager needs to retrieve some basic information about the plugin such as its name, description etc. This information is stored in `metadata.txt`.

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	boolean flag, True or False - True if this version is experimental
descontinuada	False	boolean flag, True or False, applies to the whole plugin and not just to the uploaded version
etiquetas	False	lista separada por vírgula, são permitidos espaços dentro de etiquetas 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	one of Raster, Vector, Database, Mesh and Web
dependências_de_complementos	False	PIP-like comma separated list of other plugins to install, use plugin names coming from their metadata's name field
servidor	False	boolean flag, True or False, determines if the plugin has a server interface
hasProcessing-Provider	False	boolean flag, True or False, determines if the plugin provides processing algorithms

By default, plugins are placed in the *Plugins* menu (we will see in the next section how to add a menu entry for your plugin) but they can also be placed into *Raster*, *Vector*, *Database*, *Mesh* and *Web* menus.

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 recente 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.
```

(continua na próxima página)

```

    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".
; Both "MyOtherPlugin" and "YetAnotherPlugin" names come from their own metadata's
; name field
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("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 from signal of the canvas
        self.iface.mapCanvas().renderComplete.disconnect(self.renderTest)

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

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

```

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

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

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

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

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

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("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.

While help and about actions can also be added to your custom menu, a convenient place to make them available is in the QGIS main *Help ► Plugins* menu. This is done using the `pluginHelpMenu()` method.

```
def initGui(self):

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

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

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

def unload(self):

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

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 Documenting plugins

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 Translating plugins

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

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

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

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 Sharing your plugin

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.

16.1.6 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.

Automate packaging, release and translation with qgis-plugin-ci

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

It allows releasing, translating, publishing or generating an XML plugin repository file via CLI or in CI actions.

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`.

Arquivo de Recurso

Some plugins use resource files, for example `resources.qrc` which define resources for the GUI, such as icons:

```
<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.

16.2 Partes de código

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

```
1 from qgis.core import (
2     QgsProject,
3     QgsApplication,
4     QgsMapLayer,
5 )
6
7 from qgis.gui import (
8     QgsGui,
9     QgsOptionsWidgetFactory,
10    QgsOptionsPageWidget,
11    QgsLayerTreeEmbeddedWidgetProvider,
12    QgsLayerTreeEmbeddedWidgetRegistry,
13 )
14
15 from qgis.PyQt.QtCore import Qt
16 from qgis.PyQt.QtWidgets import (
17     QMessageBox,
18     QAction,
19     QHBoxLayout,
20     QComboBox,
21 )
22 from qgis.PyQt.QtGui import QIcon
```

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")
```

It is also possible to allow users to customize key shortcuts for the provided actions. This is done by adding:

```
1 # in the initGui() function
2 QgsGui.shortcutsManager().registerAction(self.key_action)
3
4 # and in the unload() function
5 QgsGui.shortcutsManager().unregisterAction(self.key_action)
```

16.2.2 How to reuse QGIS icons

Because they are well-known and convey a clear message to the users, you may want sometimes to reuse QGIS icons in your plugin instead of drawing and setting a new one. Use the `getThemeIcon()` method.

For example, to reuse the  `mActionFileOpen.svg` icon available in the QGIS code repository:

```
1 # e.g. somewhere in the initGui
2 self.file_open_action = QAction(
3     QgsApplication.getThemeIcon("/mActionFileOpen.svg"),
4     self.tr("Select a File..."),
5     self.iface.mainWindow()
6 )
7 self.iface.addPluginToMenu("MyPlugin", self.file_open_action)
```

`iconPath()` is another method to call QGIS icons. Find examples of calls to theme icons at [QGIS embedded images - Cheatsheet](#).

16.2.3 Interface para plugin na caixa de diálogo de opções

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.

```
1 class MyPluginOptionsFactory(QgsOptionsWidgetFactory):
2
3     def __init__(self):
4         super().__init__()
```

(continua na próxima página)

(continuação da página anterior)

```
5
6     def icon(self):
7         return QIcon('icons/my_plugin_icon.svg')
8
9     def createWidget(self, parent):
10        return ConfigOptionsPage(parent)
11
12
13 class ConfigOptionsPage(QgsOptionsPageWidget):
14
15     def __init__(self, parent):
16         super().__init__(parent)
17         layout = QHBoxLayout()
18         layout.setContentsMargins(0, 0, 0, 0)
19         self.setLayout(layout)
```

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

```
1 from qgis.PyQt.QtWidgets import QHBoxLayout
2 from qgis.gui import QgsOptionsWidgetFactory, QgsOptionsPageWidget
3
4
5 class MyPlugin:
6     """QGIS Plugin Implementation."""
7
8     def __init__(self, iface):
9         """Constructor.
10
11         :param iface: An interface instance that will be passed to this class
12                       which provides the hook by which you can manipulate the QGIS
13                       application at run time.
14         :type iface: QgsInterface
15         """
16         # Save reference to the QGIS interface
17         self.iface = iface
18
19
20     def initGui(self):
21         self.options_factory = MyPluginOptionsFactory()
22         self.options_factory.setTitle(self.tr('My Plugin'))
23         iface.registerOptionsWidgetFactory(self.options_factory)
24
25     def unload(self):
26         iface.unregisterOptionsWidgetFactory(self.options_factory)
```

Dica: Add custom tabs to layer properties dialog

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

16.2.4 Embed custom widgets for layers in the layer tree

Beside usual layer symbology elements displayed next or below the layer entry in the *Layers* panel, you can add your own widgets, allowing for quick access to some actions that are often used with a layer (setup filtering, selection, style, refreshing a layer with a button widget, create a layer based time slider or just show extra layer information in a Label there, or ...). These so-called **Layer tree embedded widgets** are made available through the layer's properties *Legend* tab for individual layers.

The following code snippet creates a drop-down in the legend which shows you the layer-styles available for the layer, allowing to quickly switch between the different layer styles.

```

1 class LayerStyleComboBox(QComboBox):
2     def __init__(self, layer):
3         QComboBox.__init__(self)
4         self.layer = layer
5         for style_name in layer.styleManager().styles():
6             self.addItem(style_name)
7
8         idx = self.findText(layer.styleManager().currentStyle())
9         if idx != -1:
10            self.setCurrentIndex(idx)
11
12        self.currentIndexChanged.connect(self.on_current_changed)
13
14    def on_current_changed(self, index):
15        self.layer.styleManager().setCurrentStyle(self.itemText(index))
16
17 class LayerStyleWidgetProvider(QgsLayerTreeEmbeddedWidgetProvider):
18     def __init__(self):
19         QgsLayerTreeEmbeddedWidgetProvider.__init__(self)
20
21     def id(self):
22         return "style"
23
24     def name(self):
25         return "Layer style chooser"
26
27     def createWidget(self, layer, widgetIndex):
28         return LayerStyleComboBox(layer)
29
30     def supportsLayer(self, layer):
31         return True # any layer is fine
32
33 provider = LayerStyleWidgetProvider()
34 QgsGui.layerTreeEmbeddedWidgetRegistry().addProvider(provider)

```

Then from a given layer's *Legend* properties tab, drag the *Layer style chooser* from the *Available widgets* to *Used widgets* to enable the widget in the layer tree. Embedded widgets are ALWAYS displayed at the top of their associated layer node subitems.

If you want to use the widgets from within e.g. a plugin, you can add them like this:

```

1 layer = iface.activeLayer()
2 counter = int(layer.customProperty("embeddedWidgets/count", 0))
3 layer.setCustomProperty("embeddedWidgets/count", counter+1)
4 layer.setCustomProperty("embeddedWidgets/{}/id".format(counter), "style")
5 view = self.iface.layerTreeView()
6 view.layerTreeModel().refreshLayerLegend(view.currentLegendNode())
7 view.currentNode().setExpanded(True)

```

16.3 Configurações de IDE para gravar e depurar complementos

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.

16.3.1 Complementos úteis para escrever complementos Python

Alguns complementos são convenientes ao escrever complementos Python. De *Complementos* ► *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 um complemento.

Aviso: Despite our constant efforts, information beyond this line may not be updated for QGIS 3.

16.3.2 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`.

16.3.3 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.

16.3.4 Depurando usando Eclipse e PyDev

Instalação

Para usar o Eclipse, verifique se você instalou o seguinte

- Eclipse
- Plugin Aptana Studio 3 ou PyDev
- QGIS 3.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 lote 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
start /B 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.

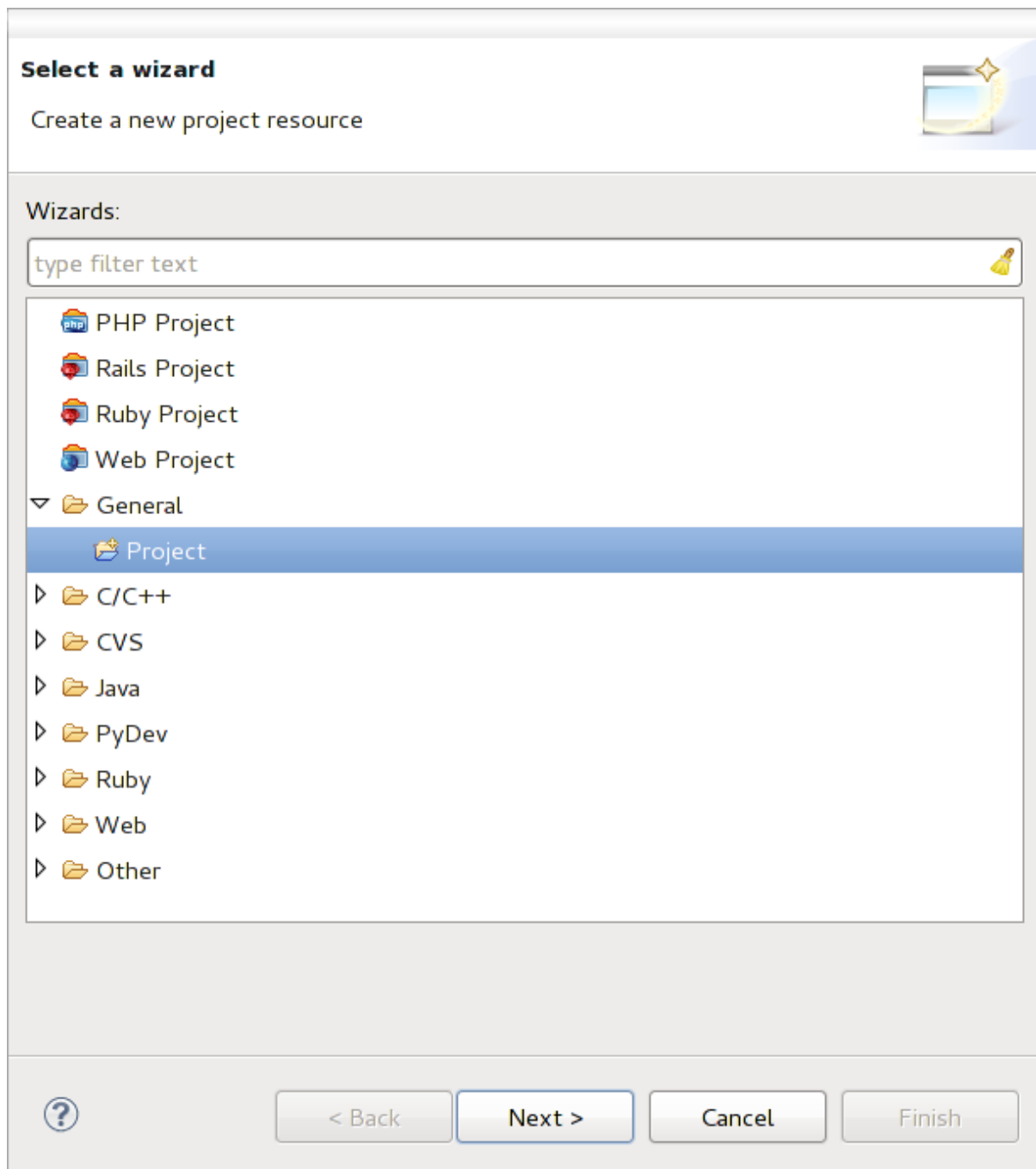


Figura16.1: Projeto Eclipse

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.

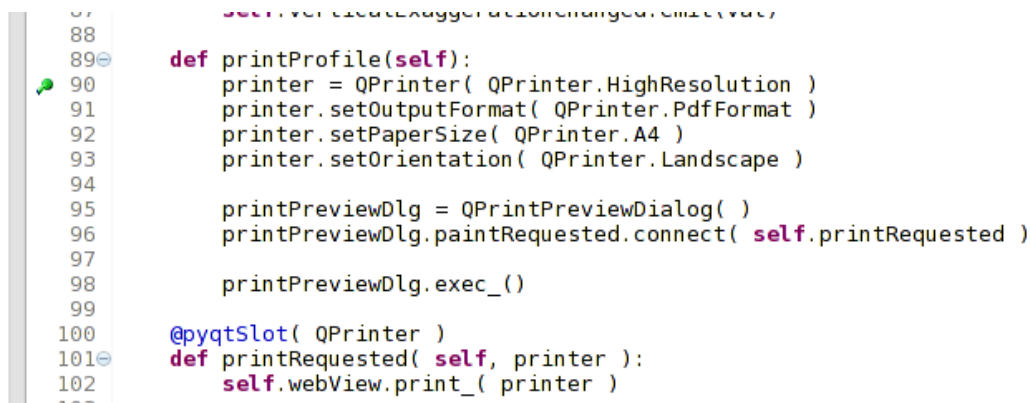
Agora, na exibição *Project Explorer*, sua árvore de fontes é exibida e você pode começar a trabalhar com o código. Você já possui destaque de sintaxe e todas as outras ferramentas IDE poderosas disponíveis.

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).



```

87         self.verticalExaggerationChanged.emit(val)
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 )

```

Figura16.2: 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.

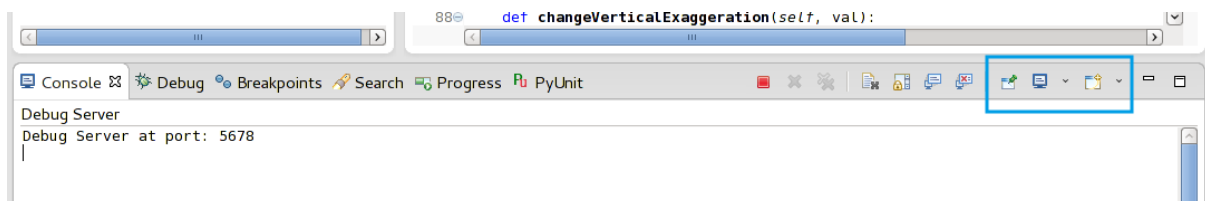


Figura16.3: Console de Depuração do PyDev

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.

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*.

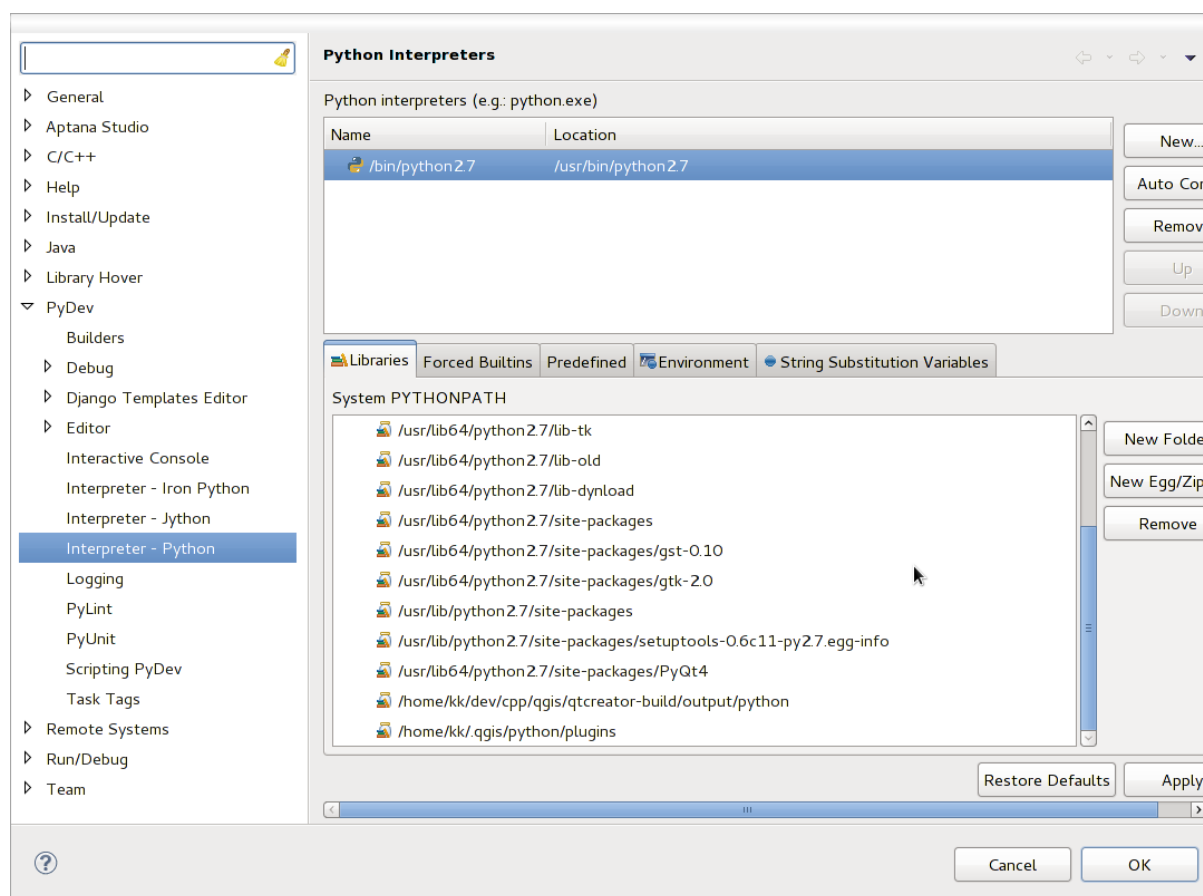


Figura16.4: Console de Depuração do PyDev

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).
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.

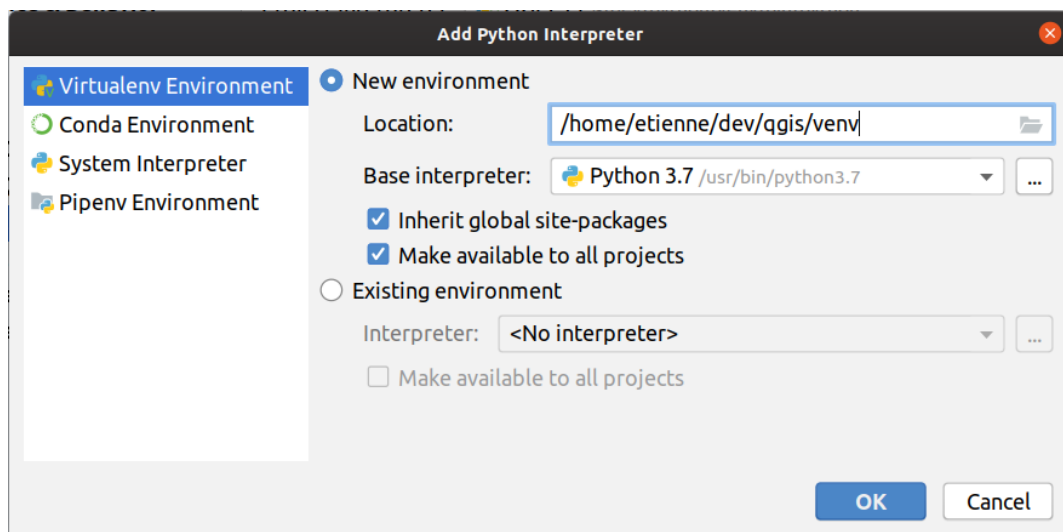
Nota: Toda vez que a API do QGIS for alterada (por exemplo, se você estiver compilando o mestre do QGIS e o arquivo SIP for alterado), volte a esta página e clique em *Aplicar*. Isso permitirá que o Eclipse analise todas as bibliotecas novamente.

16.3.5 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>

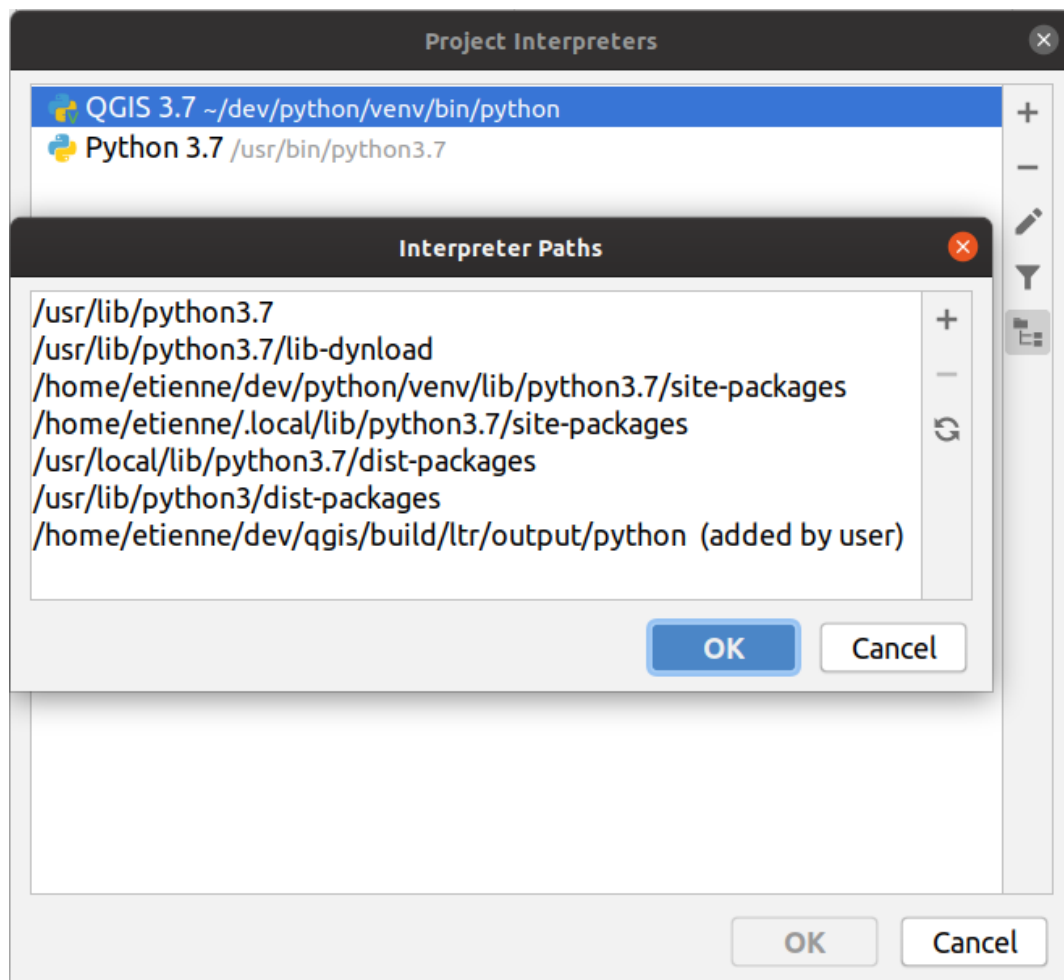
Estamos assumindo que você compilou o QGIS no Ubuntu com o diretório de compilação fornecido `~/dev/qgis/build/master`. Não é obrigatório ter um QGIS auto-compilado, mas apenas isso foi testado. Os caminhos devem ser adaptados.

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. Na nova janela, selecione seu novo intérprete QGIS e clique no último ícone no menu vertical *Mostrar caminhos para o intérprete selecionado*.
3. Finalmente, adicione o seguinte caminho absoluto à lista :file:~/dev/qgis/build/master/output/python'.



1. Reinicie PyCharm e você pode começar a usar este novo ambiente virtual Python para todos os seus 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.

Na versão profissional do PyCharm, a depuração remota está funcionando bem. Para a edição comunitária, a depuração remota não está disponível. Você só pode ter acesso a um depurador local, o que significa que o código deve rodar *dentro* do PyCharm (como script ou teste unitário), e não no próprio QGIS. Para código Python rodando *no* QGIS, você pode usar o plugin *First Aid* mencionado acima.

16.3.6 Debugging using PDB

Se você não usa uma IDE como Eclipse ou PyCharm, você pode depurar usando o PDB, seguindo estes passos.

1. Primeiro adicione este código no local onde você gostaria de depurar

```
# 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. Em seguida, execute o QGIS a partir da linha de comando.

No Linux faça:

```
$ ./Qgis
```

No macOS faça:

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

3. E quando a aplicação atingir seu ponto de parada, você pode digitar no console!

TODO:

Adicionar informações de teste

16.4 Lançando seu 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:

16.4.1 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

16.4.2 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 sub-complemento 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.

16.4.3 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
5. a license file must be included, saved as `LICENSE` with no extension (i.e. not `LICENSE.txt` for example)

Estrutura do complemento

Following the validation rules the compressed (.zip) package of your plugin must have a specific structure to validate as a functional plugin. As the plugin will be unzipped inside the users plugins folder it must have it's own directory inside the .zip file to not interfere with other plugins. Mandatory files are: `metadata.txt`, `__init__.py` and `LICENSE`. But it would be nice to have a `README` and of course an icon to represent the plugin. Following is an example of how a `plugin.zip` could look like.

```
plugin.zip
  pluginfolder/
    |-- i18n
    |   |-- translation_file_de.ts
    |-- img
    |   |-- icon.png
    |   |-- iconsource.svg
    |-- __init__.py
    |-- LICENSE
    |-- Makefile
    |-- metadata.txt
    |-- more_code.py
    |-- main_code.py
    |-- README
    |-- ui_Qt_user_interface_file.ui
```

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.

Escrevendo um complemento de processamento

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:

```

1 from qgis.core import QgsApplication
2 from .processing_provider.provider import Provider
3
4 class YourPluginName:
5
6     def __init__(self):
7         self.provider = None
8
9     def initProcessing(self):
10         self.provider = Provider()
11         QgsApplication.processingRegistry().addProvider(self.provider)
12
13     def initGui(self):
14         self.initProcessing()
15
16     def unload(self):
17         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.

```

1 from qgis.core import QgsProcessingProvider
2 from qgis.PyQt.QtGui import QIcon
3
4 from .example_processing_algorithm import ExampleProcessingAlgorithm
5
6
7 class Provider(QgsProcessingProvider):
8
9     """ The provider of our plugin. """
10
11     def loadAlgorithms(self):
12         """ Load each algorithm into the current provider. """
13         self.addAlgorithm(ExampleProcessingAlgorithm())
14         # add additional algorithms here
15         # self.addAlgorithm(MyOtherAlgorithm())
16
17     def id(self) -> str:
18         """The ID of your plugin, used for identifying the provider.
19
20         This string should be a unique, short, character only string,
21         eg "qgis" or "gdal". This string should not be localised.
22         """
23         return 'yourplugin'
24
25     def name(self) -> str:
26         """The human friendly name of your plugin in Processing.
27
28         This string should be as short as possible (e.g. "Lastools", not
29         "Lastools version 1.0.1 64-bit") and localised.
30         """
31         return self.tr('Your plugin')
32
33     def icon(self) -> QIcon:
34         """Should return a QIcon which is used for your provider inside
35         the Processing toolbox.
36         """
37         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.

You should have a tree similar to this :

```
1  └─ your_plugin_root_folder
2    ├── __init__.py
3    ├── LICENSE
4    ├── metadata.txt
5    └─ processing_provider
6        ├── example_processing_algorithm.py
7        ├── __init__.py
8        └─ provider.py
```

1. Agora você pode recarregar seu complemento no QGIS e deverá ver o seu exemplo de script na caixa de ferramentas Processar e modelador.

Usando Camadas de Complementos

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

```
1 from qgis.core import (
2     QgsPluginLayer,
3     QgsPluginLayerType,
4     QgsMapLayerRenderer,
5     QgsApplication,
6     QgsProject,
7 )
8
9 from qgis.PyQt.QtGui import QImage
```

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.

18.1 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.

```
1 class WatermarkLayerRenderer(QgsMapLayerRenderer):
2
3     def __init__(self, layerId, rendererContext):
4         super().__init__(layerId, rendererContext)
5
6     def render(self):
7         image = QImage("/usr/share/icons/hicolor/128x128/apps/qgis.png")
8         painter = self.rendererContext().painter()
9         painter.save()
10        painter.drawImage(10, 10, image)
11        painter.restore()
```

(continua na próxima página)

(continuação da página anterior)

```

12         return True
13
14 class WatermarkPluginLayer(QgsPluginLayer):
15
16     LAYER_TYPE="watermark"
17
18     def __init__(self):
19         super().__init__(WatermarkPluginLayer.LAYER_TYPE, "Watermark plugin layer")
20         self.setValid(True)
21
22     def createMapRenderer(self, rendererContext):
23         return WatermarkLayerRenderer(self.id(), rendererContext)
24
25     def setTransformContext(self, ct):
26         pass
27
28     # Methods for reading and writing specific information to the project file can
29     # also be added:
30
31     def readXml(self, node, context):
32         pass
33
34     def writeXml(self, node, doc, context):
35         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:

```

1 class WatermarkPluginLayerType(QgsPluginLayerType):
2
3     def __init__(self):
4         super().__init__(WatermarkPluginLayer.LAYER_TYPE)
5
6     def createLayer(self):
7         return WatermarkPluginLayer()
8
9     # You can also add GUI code for displaying custom information
10    # in the layer properties
11    def showLayerProperties(self, layer):
12        pass
13
14
15    # Keep a reference to the instance in Python so it won't
16    # be garbage collected
17    plt = WatermarkPluginLayerType()
18
19    assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)

```

Biblioteca de análise de rede

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

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

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.

19.1 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)

19.2 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 graph compatible with such libraries as [BGL](#) or [NetworkX](#).

To calculate edge properties the programming pattern [strategy](#) is used. For now only `QgsNetworkDistanceStrategy` (that takes into account the length of the route) and `QgsNetworkSpeedStrategy` (that also considers the speed) are available. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses a strategy that computes travel time using edge length and speed value from attributes.

Chegou a hora de mergulhar no processo.

Em primeiro lugar, para usar esta biblioteca devemos importar o módulo de análise

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

Then some examples for creating a director

```
1 # don't use information about road direction from layer attributes,
2 # all roads are treated as two-way
3 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
4     ↳QgsVectorLayerDirector.DirectionBoth)
5
6 # use field with index 5 as source of information about road direction.
7 # one-way roads with direct direction have attribute value "yes",
8 # one-way roads with reverse direction have the value "1", and accordingly
9 # bidirectional roads have "no". By default roads are treated as two-way.
10 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

```
1 director = QgsVectorLayerDirector(vectorLayer,
2     directionFieldId,
3     directDirectionValue,
4     reverseDirectionValue,
5     bothDirectionValue,
6     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

É necessário então criar uma estratégia para calcular as propriedades das bordas

```
1 # The index of the field that contains information about the edge speed
2 attributeId = 1
3 # Default speed value
4 defaultValue = 50
5 # Conversion from speed to metric units ('1' means no conversion)
6 toMetricFactor = 1
7 strategy = QgsNetworkSpeedStrategy(attributeId, defaultValue, toMetricFactor)
```

E conte ao diretor sobre essa estratégia

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', 3)
director.addStrategy(strategy)
```

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

- `crs` — coordinate reference system to use. Mandatory argument.
- `otfEnabled` — use “on the fly” reprojection or no. By default `True` (use OTF).
- `TolerânciaTopológica` — tolerância topológica. O valor padrão é `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()
```

Com o próximo código podemos obter os índices de vértices de nossos pontos

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

19.3 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.

```
1 from qgis.core import *
2 from qgis.gui import *
3 from qgis.analysis import *
4 from qgis.PyQt.QtCore import *
5 from qgis.PyQt.QtGui import *
6
7 vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
8 ↪ 'lines')
9 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
10 ↪ QgsVectorLayerDirector.DirectionBoth)
11 strategy = QgsNetworkDistanceStrategy()
12 director.addStrategy(strategy)
13 builder = QgsGraphBuilder(vectorLayer.crs())
```

(continua na próxima página)

(continuação da página anterior)

```

12
13 pStart = QgsPointXY(1179661.925139, 5419188.074362)
14 tiedPoint = director.makeGraph(builder, [pStart])
15 pStart = tiedPoint[0]
16
17 graph = builder.graph()
18
19 idStart = graph.findVertex(pStart)
20
21 tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)
22
23 i = 0
24 while (i < tree.edgeCount()):
25     rb = QgsRubberBand(iface.mapCanvas())
26     rb.setColor (Qt.red)
27     rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
28     rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
29     i = i + 1

```

Same thing but using the `dijkstra()` method

```

1 from qgis.core import *
2 from qgis.gui import *
3 from qgis.analysis import *
4 from qgis.PyQt.QtCore import *
5 from qgis.PyQt.QtGui import *
6
7 vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
8                               ↪ 'lines')
9
10 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
11                                   ↪ QgsVectorLayerDirector.DirectionBoth)
12 strategy = QgsNetworkDistanceStrategy()
13 director.addStrategy(strategy)
14 builder = QgsGraphBuilder(vectorLayer.crs())
15
16 pStart = QgsPointXY(1179661.925139, 5419188.074362)
17 tiedPoint = director.makeGraph(builder, [pStart])
18 pStart = tiedPoint[0]
19
20 graph = builder.graph()
21
22 idStart = graph.findVertex(pStart)
23
24 (tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)
25
26 for edgeId in tree:
27     if edgeId == -1:
28         continue
29     rb = QgsRubberBand(iface.mapCanvas())
30     rb.setColor (Qt.red)
31     rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
32     rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())

```

19.3.1 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:

```

1 assign T = B
2 while T != A
3     add point T to path
4     get incoming edge for point T
5     look for point TT, that is start point of this edge
6     assign T = TT
7 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

```

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',
9                               ↪ 'lines')
10 builder = QgsGraphBuilder(vectorLayer.sourceCrs())
11 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
12                                   ↪ QgsVectorLayerDirector.DirectionBoth)
13 strategy = QgsNetworkDistanceStrategy()
14 director.addStrategy(strategy)
15
16 startPoint = QgsPointXY(1179661.925139, 5419188.074362)
17 endPoint = QgsPointXY(1180942.970617, 5420040.097560)
18
19 tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
20 tStart, tStop = tiedPoints
21
22 graph = builder.graph()
23 idxStart = graph.findVertex(tStart)
24
25 tree = QgsGraphAnalyzer.shortestTree(graph, idxStart, 0)
26
27 idxStart = tree.findVertex(tStart)
28 idxEnd = tree.findVertex(tStop)
29
30 if idxEnd == -1:
31     raise Exception('No route!')
32
33 # Add last point
34 route = [tree.vertex(idxEnd).point()]
35
36 # Iterate the graph
37 while idxEnd != idxStart:
38     edgeIds = tree.vertex(idxEnd).incomingEdges()
39     if len(edgeIds) == 0:
40         break
41     edge = tree.edge(edgeIds[0])

```

(continua na próxima página)

(continuação da página anterior)

```

40     route.insert(0, tree.vertex(edge.fromVertex()).point())
41     idxEnd = edge.fromVertex()
42
43     # Display
44     rb = QgsRubberBand(iface.mapCanvas())
45     rb.setColor(Qt.green)
46
47     # This may require coordinate transformation if project's CRS
48     # is different than layer's CRS
49     for p in route:
50         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',
9                               ↪ 'lines')
10 director = QgsVectorLayerDirector(vectorLayer, -1, ' ', ' ', ' ',
11                                   ↪ QgsVectorLayerDirector.DirectionBoth)
12 strategy = QgsNetworkDistanceStrategy()
13 director.addStrategy(strategy)
14
15 builder = QgsGraphBuilder(vectorLayer.sourceCrs())
16
17 startPoint = QgsPointXY(1179661.925139, 5419188.074362)
18 endPoint = QgsPointXY(1180942.970617, 5420040.097560)
19
20 tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
21 tStart, tStop = tiedPoints
22
23 graph = builder.graph()
24 idxStart = graph.findVertex(tStart)
25 idxEnd = graph.findVertex(tStop)
26
27 (tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idxStart, 0)
28
29 if tree[idxEnd] == -1:
30     raise Exception('No route!')
31
32 # Total cost
33 cost = costs[idxEnd]
34
35 # Add last point
36 route = [graph.vertex(idxEnd).point()]
37
38 # Iterate the graph
39 while idxEnd != idxStart:
40     idxEnd = graph.edge(tree[idxEnd]).fromVertex()
41     route.insert(0, graph.vertex(idxEnd).point())
42
43 # Display
44 rb = QgsRubberBand(iface.mapCanvas())
45 rb.setColor(Qt.red)
46
47 # This may require coordinate transformation if project's CRS

```

(continua na próxima página)

```

46 # is different than layer's CRS
47 for p in route:
48     rb.addPoint(p)

```

19.3.2 Áreas de disponibilidade

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 than 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

```

1 director = QgsVectorLayerDirector(vectorLayer, -1, ' ', ' ', ' ',
2   ↳QgsVectorLayerDirector.DirectionBoth)
3 strategy = QgsNetworkDistanceStrategy()
4 director.addStrategy(strategy)
5 builder = QgsGraphBuilder(vectorLayer.crs())
6
7 pStart = QgsPointXY(1179661.925139, 5419188.074362)
8 delta = iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1
9
10 rb = QgsRubberBand(iface.mapCanvas())
11 rb.setColor(Qt.green)
12 rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
13 rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
14 rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
15 rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() + delta))
16
17 tiedPoints = director.makeGraph(builder, [pStart])
18 graph = builder.graph()
19 tStart = tiedPoints[0]
20
21 idStart = graph.findVertex(tStart)
22
23 (tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)
24
25 upperBound = []
26 r = 1500.0
27 i = 0
28 tree.reverse()
29
30 while i < len(cost):
31     if cost[i] > r and tree[i] != -1:
32         outVertexId = graph.edge(tree[i]).toVertex()
33         if cost[outVertexId] < r:
34             upperBound.append(i)
35     i = i + 1
36

```

(continua na próxima página)

(continuação da página anterior)

```
37 for i in upperBound:
38     centerPoint = graph.vertex(i).point()
39     rb = QgsRubberBand(iface.mapCanvas())
40     rb.setColor(Qt.red)
41     rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() - delta))
42     rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() - delta))
43     rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() + delta))
44     rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() + delta))
```


20.1 Introdução

Para saber mais sobre o Servidor QGIS, leia o:ref:QGIS-Server-manual.

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. QGIS Server FCGI: a FCGI binary application `qgis_mapserv.fcgi` that together with a web server implements a set of OGC services (WMS, WFS, WCS etc.) and OGC APIs (WFS3/OAPIF)
3. QGIS Development Server: a development server binary application `qgis_mapserver` that implements a set of OGC services (WMS, WFS, WCS etc.) and OGC APIs (WFS3/OAPIF)

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 → Run QGIS Server as a standalone WSGI/HTTP service
- FILTERS → Aprimora/personaliza o Servidor QGIS com complementos de filtro
- SERVICES → Adiciona um novo *SERVICE*
- OGC APIs → Adiciona um novo *OGC API*

Embedding and standalone applications require using the QGIS Server Python API directly from another Python script or application. The remaining options are better suited for when you want to add custom features to a standard QGIS Server binary application (FCGI or development server): in this case you'll need to write a Python plugin for the server application and register your custom filters, services or APIs.

20.2 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)
- `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:

```

1 initialize the QgsApplication
2 create the QgsServer
3 the main server loop waits forever for client requests:
4     for each incoming request:
5         create a QgsServerRequest request
6         create a QgsServerResponse response
7         call QgsServer.handleRequest(request, response)
8             filter plugins may be executed
9         send the output to the client

```

Inside the `QgsServer.handleRequest(request, response)` method the filter plugins callbacks are called and `QgsServerRequest` and `QgsServerResponse` are made available to the plugins through the `QgsServerInterface` class.

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.

20.3 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:

```

1 from qgis.core import QgsApplication
2 from qgis.server import *
3 app = QgsApplication([], False)
4
5 # Create the server instance, it may be a single one that
6 # is reused on multiple requests
7 server = QgsServer()
8
9 # Create the request by specifying the full URL and an optional body
10 # (for example for POST requests)
11 request = QgsBufferServerRequest(
12     'http://localhost:8081/?MAP=/qgis-server/projects/helloworld.qgs' +
13     '&SERVICE=WMS&REQUEST=GetCapabilities')
14
15 # Create a response objects
16 response = QgsBufferServerResponse()
17
18 # Handle the request
19 server.handleRequest(request, response)
20
21 print(response.headers())

```

(continua na próxima página)

(continuação da página anterior)

```

22 print(response.body().data().decode('utf8'))
23
24 app.exitQgis()

```

Here is a complete standalone application example developed for the continuous integrations testing on QGIS source code repository, it showcases a wide set of different plugin filters and authentication schemes (not mean for production because they were developed for testing purposes only but still interesting for learning): [qgis_wrapped_server.py](#)

20.4 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.

20.4.1 Complementos de filtro de servidor

Os filtros vêm em três sabores diferentes e podem ser instanciados subclassificando uma das classes abaixo e chamando o método correspondente de `QgsServerInterface`:

Tipo de Filtro	Classe Básica	Registro da QgsServerInterface
I/O	<code>QgsServerFilter</code>	<code>registerFilter()</code>
Controle de Acesso	<code>QgsAccessControlFilter</code>	<code>registerAccessControl()</code>
Cache	<code>QgsServerCacheFilter</code>	<code>registerServerCache()</code>

Filtros I/O

I/O filters can modify the server input and output (the request and the response) of the core services (WMS, WFS etc.) allowing to do any kind of manipulation of the services workflow. It is possible for example to restrict the access to selected layers, to inject an XSL stylesheet to the XML response, to add a watermark to a generated WMS image and so on.

From this point, you might find useful a quick look to the [server plugins API docs](#).

Cada filtro deve implementar pelo menos um dos três retornos de chamada:

- `onRequestReady()`
- `onResponseComplete()`
- `onSendResponse()`

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).

All these methods return a boolean value indicating if the call should be propagated to the subsequent filters. If one of these method returns `False` then the chain stop, otherwise the call will propagate to the next filter.

Here is the pseudo code showing how the server handles a typical request and when the filter's callbacks are called:

```
1 for each incoming request:
2     create GET/POST request handler
3     pass request to an instance of QgsServerInterface
4     call onRequestReady filters
5
6     if there is not a response:
7         if SERVICE is WMS/WFS/WCS:
8             create WMS/WFS/WCS service
9             call service's executeRequest
10            possibly call onSendResponse for each chunk of bytes
11            sent to the client by a streaming services (WFS)
12        call onResponseComplete
13    request handler sends the response to the client
```

Os parágrafos seguintes descrevem em detalhes as funções de retorno disponíveis.

onRequestReady

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
- redireciona
- adicionar/remover certos parâmetros (nomes de tipo, por exemplo)
- levantar exceções

Você pode até substituir completamente um serviço principal alterando o parâmetro **SERVIÇO** e, portanto, ignorar completamente o serviço principal (não que isso faça muito sentido).

onSendResponse

This is called whenever any partial output is flushed from response buffer (i.e to **FCGI** stdout if the fcgi server is used) and from there, to the client. This occurs when huge content is streamed (like WFS GetFeature). In this case `onSendResponse()` may be called multiple times.

Note that if the response is not streamed, then `onSendResponse()` will not be called at all.

In all case, the last (or unique) chunk will be sent to client after a call to `onResponseComplete()`.

Returning `False` will prevent flushing of data to the client. This is desirable when a plugin wants to collect all chunks from a response and examine or change the response in `onResponseComplete()`.

onResponseComplete

This is called once when core services (if hit) finish their process and the request is ready to be sent to the client. As discussed above, this method will be called before the last (or unique) chunk of data is sent to the client. For streaming services, multiple calls to `onSendResponse()` might have been called.

`onResponseComplete()` is the ideal place to provide new services implementation (WPS or custom services) and to perform direct manipulation of the output coming from core services (for example to add a watermark upon a WMS image).

Note that returning `False` will prevent the next plugins to execute `onResponseComplete()` but, in any case, prevent response to be sent to the client.

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 `qgis3-server-vagrant` example plugin discussed here (with many more) is available on github, a few server plugins are also published in the official [QGIS plugins repository](#).

Arquivos de complementos

Aqui está a estrutura de diretórios do nosso exemplo de plugin de servidor.

```
1 PYTHON_PLUGINS_PATH/
2   HelloServer/
3     __init__.py    --> *required*
4     HelloServer.py --> *required*
5     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:

- `onRequestReady()`
- `onResponseComplete()`
- `onSendResponse()`

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

```

1 class HelloFilter(QgsServerFilter):
2
3     def __init__(self, serverIface):
4         super().__init__(serverIface)
5
6     def onRequestReady(self) -> bool:
7         QgsMessageLog.logMessage("HelloFilter.onRequestReady")
8         return True
9
10    def onSendResponse(self) -> bool:
11        QgsMessageLog.logMessage("HelloFilter.onSendResponse")
12        return True
13
14    def onResponseComplete(self) -> bool:
15        QgsMessageLog.logMessage("HelloFilter.onResponseComplete")
16        request = self.serverInterface().requestHandler()
17        params = request.parameterMap()
18        if params.get('SERVICE', '').upper() == 'HELLO':
19            request.clear()
20            request.setResponseHeader('Content-type', 'text/plain')
21            # Note that the content is of type "bytes"
22            request.appendBody(b'HelloServer!')
23        return True

```

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

```

class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(serverIface), 100)

```

The second parameter of `registerFilter()` sets a priority which defines the order for the callbacks with the same name (the lower priority is invoked first).

By using the three callbacks, plugins can manipulate the input and/or the output of the server in many different ways. In every moment, the plugin instance has access to the `QgsRequestHandler` through the `QgsServerInterface`. The `QgsRequestHandler` class has plenty of methods that can be used to alter the input parameters before entering the core processing of the server (by using `requestReady()`) or after the request has been processed by the core services (by using `sendResponse()`).

Os exemplos a seguir cobrem alguns casos de uso comuns:

Modificando a entrada

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:

```

1 class ParamsFilter(QgsServerFilter):
2
3     def __init__(self, serverIface):
4         super(ParamsFilter, self).__init__(serverIface)
5
6     def onRequestReady(self) -> bool:
7         request = self.serverInterface().requestHandler()
8         params = request.parameterMap()
9         request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')
10        return True
11
12    def onResponseComplete(self) -> bool:
13        request = self.serverInterface().requestHandler()
14        params = request.parameterMap()
15        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
16            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.onResponseComplete")
17        else:
18            QgsMessageLog.logMessage("FAIL - ParamsFilter.onResponseComplete")
19        return True

```

Este é um extrato do que você vê no arquivo de registro:

```

1 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]
↳HelloServerServer - loading filter ParamsFilter
2 src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0]
↳Server plugin HelloServer loaded!
3 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0]
↳Server python plugins loaded
4 src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms]
↳inserting pair SERVICE // HELLO into the parameter map
5 src/mapserver/qgsserverfilter.cpp: 42: (onRequestReady) [0ms] QgsServerFilter
↳plugin default onRequestReady called
6 src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]
↳SUCCESS - ParamsFilter.onResponseComplete

```

On the highlighted line the “SUCCESS” string indicates that the plugin passed the test.

The same technique can be exploited to use a custom service instead of a core one: you could for example skip a **WFS SERVICE** request or any other core request just by changing the **SERVICE** parameter to something different and the core service will be skipped. Then you can inject your custom results into the output and send them to the client (this is explained below).

Dica: Se você realmente deseja implementar um serviço personalizado, é recomendado subclasse `QgsService` e registrar seu serviço em `registerFilter()` chamando seu `registerService(service)`

Modificando ou substituindo a saída

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:

```

1  from qgis.server import *
2  from qgis.PyQt.QtCore import *
3  from qgis.PyQt.QtGui import *
4
5  class WatermarkFilter(QgsServerFilter):
6
7      def __init__(self, serverIface):
8          super().__init__(serverIface)
9
10     def onResponseComplete(self) -> bool:
11         request = self.serverInterface().requestHandler()
12         params = request.parameterMap( )
13         # Do some checks
14         if (params.get('SERVICE').upper() == 'WMS' \
15             and params.get('REQUEST').upper() == 'GETMAP' \
16             and not request.exceptionRaised() ):
17             QgsMessageLog.logMessage("WatermarkFilter.onResponseComplete: image_
↪ready %s" % request.parameter("FORMAT"))
18             # Get the image
19             img = QImage()
20             img.loadFromData(request.body())
21             # Adds the watermark
22             watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/
↪watermark.png'))
23             p = QPainter(img)
24             p.drawImage(QRect( 20, 20, 40, 40), watermark)
25             p.end()
26             ba = QByteArray()
27             buffer = QBuffer(ba)
28             buffer.open(QIODevice.WriteOnly)
29             img.save(buffer, "PNG" if "png" in request.parameter("FORMAT") else
↪"JPG")
30             # Set the body
31             request.clearBody()
32             request.appendBody(ba)
33             return True

```

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.

Filtros de controle de acesso

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

Aqui está a estrutura do diretório do nosso exemplo de plugin:

```
1 PYTHON_PLUGINS_PATH/
2   MyAccessControl/
3     __init__.py    --> *required*
4     AccessControl.py --> *required*
5     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`

```
1 class AccessControlFilter(QgsAccessControlFilter):
2
3     def __init__(self, server_iface):
4         super().__init__(server_iface)
5
6     def layerFilterExpression(self, layer):
7         """ Return an additional expression filter """
8         return super().layerFilterExpression(layer)
9
10    def layerFilterSubsetString(self, layer):
11        """ Return an additional subset string (typically SQL) filter """
12        return super().layerFilterSubsetString(layer)
13
14    def layerPermissions(self, layer):
15        """ Return the layer rights """
16        return super().layerPermissions(layer)
17
18    def authorizedLayerAttributes(self, layer, attributes):
19        """ Return the authorised layer attributes """
20        return super().authorizedLayerAttributes(layer, attributes)
21
22    def allowToEdit(self, layer, feature):
23        """ Are we authorised to modify the following geometry """
24        return super().allowToEdit(layer, feature)
25
26    def cacheKey(self):
27        return super().cacheKey()
28
29 class AccessControlServer:
30
31     def __init__(self, serverIface):
```

(continua na próxima página)

(continuação da página anterior)

```
32 """ Register AccessControlFilter """
33 serverIface.registerAccessControl (AccessControlFilter (serverIface), 100)
```

Este exemplo dá acesso total para todos.

É papel do complemento saber quem está conectado.

Em todos esses métodos, temos a camada no argumento para poder personalizar a restrição por camada.

layerFilterExpression

Used to add an Expression to limit the results.

For example, to limit to features where the attribute `role` is equal to `user`.

```
def layerFilterExpression(self, layer):
    return "$role = 'user'"
```

layerFilterSubsetString

Same than the previous but use the `SubsetString` (executed in the database)

For example, to limit to features where the attribute `role` is equal to `user`.

```
def layerFilterSubsetString(self, layer):
    return "role = 'user'"
```

layerPermissions

Limitar o acesso à camada.

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.

For example, to limit everything on read only access:

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

authorizedLayerAttributes

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

The argument attribute return the current set of visible attributes.

For example, to hide the `role` attribute:

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

allowToEdit

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

It is used in the WFS-Transaction protocol.

For example, to be able to edit only feature that has the attribute `role` with the value `user`:

```
def allowToEdit(self, layer, feature):
    return feature.attribute('role') == 'user'
```

cacheKey

QGIS Server maintains 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.

20.4.2 Serviços personalizados

In QGIS Server, core services such as WMS, WFS and WCS are implemented as subclasses of `QgsService`.

To implement a new service that will be executed when the query string parameter `SERVICE` matches the service name, you can implement 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`:

```
1 from qgis.server import QgsService
2 from qgis.core import QgsMessageLog
3
4 class CustomServiceService(QgsService):
5
6     def __init__(self):
7         QgsService.__init__(self)
8
9     def name(self):
10        return "CUSTOM"
11
12    def version(self):
13        return "1.0.0"
14
15    def executeRequest(self, request, response, project):
16        response.setStatusCode(200)
17        QgsMessageLog.logMessage('Custom service executeRequest')
18        response.write("Custom service executeRequest")
19
20
21 class CustomService():
22
```

(continua na próxima página)

```

23 def __init__(self, serverIface):
24     serverIface.serviceRegistry().registerService(CustomServiceService())

```

20.4.3 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 implement a new API that will be executed when the url path matches a certain URL, you can implement 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`:

```

1  import json
2  import os
3
4  from qgis.PyQt.QtCore import QBuffer, QIODevice, QTextStream, QRegularExpression
5  from qgis.server import (
6      QgsServiceRegistry,
7      QgsService,
8      QgsServerFilter,
9      QgsServerOgcApi,
10     QgsServerQueryStringParameter,
11     QgsServerOgcApiHandler,
12 )
13
14 from qgis.core import (
15     QgsMessageLog,
16     QgsJsonExporter,
17     QgsCircle,
18     QgsFeature,
19     QgsPoint,
20     QgsGeometry,
21 )
22
23
24 class CustomApiHandler(QgsServerOgcApiHandler):
25
26     def __init__(self):
27         super(CustomApiHandler, self).__init__()
28         self.setContentTypes([QgsServerOgcApi.HTML, QgsServerOgcApi.JSON])
29
30     def path(self):
31         return QRegularExpression("/customapi")
32
33     def operationId(self):
34         return "CustomApiXYCircle"
35
36     def summary(self):
37         return "Creates a circle around a point"
38
39     def description(self):
40         return "Creates a circle around a point"
41
42     def linkTitle(self):
43         return "Custom Api XY Circle"
44
45     def linkType(self):

```

(continua na próxima página)

(continuação da página anterior)

```

46     return QgsServerOgcApi.data
47
48     def handleRequest(self, context):
49         """Simple Circle"""
50
51         values = self.values(context)
52         x = values['x']
53         y = values['y']
54         r = values['r']
55         f = QgsFeature()
56         f.setAttributes([x, y, r])
57         f.setGeometry(QgsCircle(QgsPoint(x, y), r).toCircularString())
58         exporter = QgsJsonExporter()
59         self.write(json.loads(exporter.exportFeature(f)), context)
60
61     def templatePath(self, context):
62         # The template path is used to serve HTML content
63         return os.path.join(os.path.dirname(__file__), 'circle.html')
64
65     def parameters(self, context):
66         return [QgsServerQueryStringParameter('x', True,
↪QgsServerQueryStringParameter.Type.Double, 'X coordinate'),
67                 QgsServerQueryStringParameter(
68                     'y', True, QgsServerQueryStringParameter.Type.Double, 'Y
↪coordinate'),
69                 QgsServerQueryStringParameter('r', True,
↪QgsServerQueryStringParameter.Type.Double, 'radius')]
70
71
72 class CustomApi():
73
74     def __init__(self, serverIface):
75         api = QgsServerOgcApi(serverIface, '/customapi',
76                               'custom api', 'a custom api', '1.1')
77         handler = CustomApiHandler()
78         api.registerHandler(handler)
79         serverIface.serviceRegistry().registerApi(api)

```


Folha de dicas para PyQGIS

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

```
1 from qgis.PyQt.QtCore import (  
2     QRectF,  
3 )  
4  
5 from qgis.core import (  
6     Qgis,  
7     QgsProject,  
8     QgsLayerTreeModel,  
9 )  
10  
11 from qgis.gui import (  
12     QgsLayerTreeView,  
13 )
```

21.1 Interface de usuário

Alterar aparência

```
1 from qgis.PyQt.QtWidgets import QApplication  
2  
3 app = QApplication.instance()  
4 app.setStyleSheet(".QWidget {color: blue; background-color: yellow;}")  
5 # You can even read the stylesheet from a file  
6 with open("testdata/file.qss") as qss_file_content:  
7     app.setStyleSheet(qss_file_content.read())
```

Alterar ícone e título

```
1 from qgis.PyQt.QtGui import QIcon  
2  
3 icon = QIcon("/path/to/logo/file.png")
```

(continua na próxima página)

(continuação da página anterior)

```
4 iface.mainWindow().setWindowIcon(icon)
5 iface.mainWindow().setWindowTitle("My QGIS")
```

21.2 Configurações

Get QgsSettings list

```
1 from qgis.core import QgsSettings
2
3 qs = QgsSettings()
4
5 for k in sorted(qs.allKeys()):
6     print(k)
```

21.3 Barra de ferramentas

Remover barra de ferramenas

```
1 toolbar = iface.helpToolBar()
2 parent = toolbar.parentWidget()
3 parent.removeToolBar(toolbar)
4
5 # and add again
6 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])
```

21.4 Menus

Remover menu

```
1 # for example Help Menu
2 menu = iface.helpMenu()
3 menubar = menu.parentWidget()
4 menubar.removeAction(menu.menuAction())
5
6 # and add again
7 menubar.addAction(menu.menuAction())
```


21.5 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)
```

21.6 Camadas

Adicionar camada vetorial

```
layer = iface.addVectorLayer("testdata/data/data.gpkg|layername=airports",
    ↳ "Airports layer", "ogr")
if not layer or not layer.isValid():
    print("Layer failed to load!")
```

Obter camada ativa

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

Lista todas as camadas

```
from qgis.core import QgsProject

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

Obtém os nomes das camadas

```
1 from qgis.core import QgsVectorLayer
2 layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
3 QgsProject.instance().addMapLayer(layer)
4
5 layers_names = []
6 for layer in QgsProject.instance().mapLayers().values():
7     layers_names.append(layer.name())
8
9 print("layers TOC = {}".format(layers_names))
```

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

Por outro lado

```
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() # 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

```
1 from qgis.core import QgsProject
2
3 layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
4 # Set seconds (5 seconds)
5 layer.setAutoRefreshInterval(5000)
6 # Enable data reloading
7 layer.setAutoRefreshMode(Qgis.AutoRefreshMode.ReloadData)
```

Mostrar métodos

```
dir(layer)
```

Adding new feature with feature form

```
1 from qgis.core import QgsFeature, QgsGeometry
2
3 feat = QgsFeature()
4 geom = QgsGeometry()
5 feat.setGeometry(geom)
6 feat.setFields(layer.fields())
7
8 iface.openFeatureForm(layer, feat, False)
```

Adding new feature without feature form

```
1 from qgis.core import QgsGeometry, QgsPointXY, QgsFeature
2
3 pr = layer.dataProvider()
4 feat = QgsFeature()
5 feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
6 pr.addFeatures([feat])
```

Obter recursos

```
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)
```

Crie uma camada de memória a partir de IDs de recursos selecionados

```
from qgis.core import QgsFeatureRequest

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

Obter geometria

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

```
10.000000, 10.000000
```

Mover geometria

```
1 from qgis.core import QgsFeature, QgsGeometry
2 poly = QgsFeature()
3 geom = QgsGeometry.fromWkt("POINT(7 45)")
4 geom.translate(1, 1)
5 poly.setGeometry(geom)
6 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

```
1 from qgis.core import QgsProject
2
3 for layer in QgsProject.instance().mapLayers().values():
4     crs = layer.crs().authid()
5     layer.setName('{} {}'.format(layer.name(), crs))
```

Ocultar uma coluna de campo

```
1 from qgis.core import QgsEditorWidgetSetup
2
3 def fieldVisibility (layer, fname):
4     setup = QgsEditorWidgetSetup('Hidden', {})
5     for i, column in enumerate(layer.fields()):
6         if column.name() == fname:
7             layer.setEditorWidgetSetup(idx, setup)
8             break
```

(continua na próxima página)

```

9     else:
10         continue

```

Camada de WKT

```

1 from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject
2
3 layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
4 pr = layer.dataProvider()
5 poly = QgsFeature()
6 geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.09 34.89,-88.39 30.34,-89.
  ↳57 30.18,-89.73 31,-91.63 30.99,-90.87 32.37,-91.23 33.44,-90.93 34.23,-90.30 34.
  ↳99,-88.82 34.99)) ")
7 poly.setGeometry(geom)
8 pr.addFeatures([poly])
9 layer.updateExtents()
10 QgsProject.instance().addMapLayers([layer])

```

Carregar todas as camadas vetoriais do GeoPackage

```

1 from qgis.core import QgsDataProvider
2
3 fileName = "testdata/sublayers.gpkg"
4 layer = QgsVectorLayer(fileName, "test", "ogr")
5 subLayers = layer.dataProvider().subLayers()
6
7 for subLayer in subLayers:
8     name = subLayer.split(QgsDataProvider.SUBLAYER_SEPARATOR)[1]
9     uri = "%s|layername=%s" % (fileName, name,)
10     # Create layer
11     sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
12     # Add layer to map
13     QgsProject.instance().addMapLayer(sub_vlayer)

```

Carregar camada de mosaico (camada XYZ)

```

1 from qgis.core import QgsRasterLayer, QgsProject
2
3 def loadXYZ(url, name):
4     rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
5     QgsProject.instance().addMapLayer(rasterLyr)
6
7 urlWithParams = 'https://tile.openstreetmap.org/%7Bz%7D/%7Bx%7D/%7By%7D.png&
  ↳zmax=19&zmin=0&crs=EPSG3857'
8 loadXYZ(urlWithParams, 'OpenStreetMap')

```

Remover todas as camadas

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

Remover tudo

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

21.7 Tabela de conteúdos

Acessar camadas verificadas

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

Remove contextual menu

```
1 ltv = iface.layerTreeView()
2 mp = ltv.menuProvider()
3 ltv.setMenuProvider(None)
4 # Restore
5 ltv.setMenuProvider(mp)
```

21.8 Advanced TOC

Root node

```
1 from qgis.core import QgsVectorLayer, QgsProject, QgsLayerTreeLayer
2
3 root = QgsProject.instance().layerTreeRoot()
4 node_group = root.addGroup("My Group")
5
6 layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
7 QgsProject.instance().addMapLayer(layer, False)
8
9 node_group.addLayer(layer)
10
11 print(root)
12 print(root.children())
```

Access the first child node

```
1 from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer, QgsLayerTree
2
3 child0 = root.children()[0]
4 print (child0.name())
5 print (type(child0))
6 print (isinstance(child0, QgsLayerTreeLayer))
7 print (isinstance(child0.parent(), QgsLayerTree))
```

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

Encontrar grupos e nós

```
1 from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer
2
3 def get_group_layers(group):
4     print(' - group: ' + group.name())
5     for child in group.children():
6         if isinstance(child, QgsLayerTreeGroup):
7             # Recursive call to get nested groups
8             get_group_layers(child)
9         else:
10            print(' - layer: ' + child.name())
```

(continua na próxima página)

(continuação da página anterior)

```

11
12
13 root = QgsProject.instance().layerTreeRoot()
14 for child in root.children():
15     if isinstance(child, QgsLayerTreeGroup):
16         get_group_layers(child)
17     elif isinstance(child, QgsLayerTreeLayer):
18         print ('- layer: ' + child.name())

```

```

- group: My Group
- layer: layer name you like

```

Encontrar grupo pelo nome

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

```
<QgsLayerTreeGroup: My Group>
```

Encontrar camada por id

```
print(root.findLayer(layer.id()))
```

```
<QgsLayerTreeLayer: layer name you like>
```

Adicionar camada

```

1 from qgis.core import QgsVectorLayer, QgsProject
2
3 layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
4 QgsProject.instance().addMapLayer(layer1, False)
5 node_layer1 = root.addLayer(layer1)
6 # Remove it
7 QgsProject.instance().removeMapLayer(layer1)

```

Adicionar grupo

```

1 from qgis.core import QgsLayerTreeGroup
2
3 node_group2 = QgsLayerTreeGroup("Group 2")
4 root.addChildNode(node_group2)
5 QgsProject.instance().mapLayersByName("layer name you like")[0]

```

Mover camada carregada

```

1 layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
2 root = QgsProject.instance().layerTreeRoot()
3
4 myLayer = root.findLayer(layer.id())
5 myClone = myLayer.clone()
6 parent = myLayer.parent()
7
8 myGroup = root.findGroup("My Group")
9 # Insert in first position
10 myGroup.insertChildNode(0, myClone)
11
12 parent.removeChildNode(myLayer)

```

Mover camada carregada para um grupo específico

```

1 QgsProject.instance().addMapLayer(layer, False)
2
3 root = QgsProject.instance().layerTreeRoot()
4 myGroup = root.findGroup("My Group")
5 myOriginalLayer = root.findLayer(layer.id())
6 myLayer = myOriginalLayer.clone()
7 myGroup.insertChildNode(0, myLayer)
8 parent.removeChildNode(myOriginalLayer)

```

Toggle active layer visibility

```

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

```

O grupo está seleccionado

```

1 def isMyGroupSelected( groupName ):
2     myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
3     return myGroup in iface.layerTreeView().selectedNodes()
4
5 print(isMyGroupSelected( 'my group name' ))

```

```
False
```

Expand node

```

print(myGroup.isExpanded())
myGroup.setExpanded(False)

```

Hidden node trick

```

1 from qgis.core import QgsProject
2
3 model = iface.layerTreeView().layerTreeModel()
4 ltv = iface.layerTreeView()
5 root = QgsProject.instance().layerTreeRoot()
6
7 layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
8 node = root.findLayer(layer.id())
9
10 index = model.node2index( node )
11 ltv.setRowHidden( index.row(), index.parent(), True )
12 node.setCustomProperty( 'nodeHidden', 'true' )
13 ltv.setCurrentIndex(model.node2index(root))

```

Node signals

```

1 def onWillAddChildren(node, indexFrom, indexTo):
2     print ("WILL ADD", node, indexFrom, indexTo)
3
4 def onAddedChildren(node, indexFrom, indexTo):
5     print ("ADDED", node, indexFrom, indexTo)
6
7 root.willAddChildren.connect(onWillAddChildren)
8 root.addedChildren.connect(onAddedChildren)

```

Remover camada

```
root.removeLayer(layer)
```

Remover grupo

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

Criar novo sumário (TOC)

```
1 root = QgsProject.instance().layerTreeRoot()
2 model = QgsLayerTreeModel(root)
3 view = QgsLayerTreeView()
4 view.setModel(model)
5 view.show()
```

Mover nó

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

Renomear nó

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

21.9 Algoritmos de processamento

Get algorithms list

```
1 from qgis.core import QgsApplication
2
3 for alg in QgsApplication.processingRegistry().algorithms():
4     if 'buffer' == alg.name():
5         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

Para este exemplo, o resultado é armazenado em uma camada de memória temporária que é adicionada ao projeto.

```
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'}
```

Quantos algoritmos existem?

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

Quantos provedores existem?


```
from qgis.core import QgsApplication

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

Quantas expressões existem?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

21.10 Decoradores

Direito autoral

```
1 from qgis.PyQt.Qt import QTextDocument
2 from qgis.PyQt.QtGui import QFont
3
4 mQFont = "Sans Serif"
5 mQFontsize = 9
6 mLabelQString = "© QGIS 2019"
7 mMarginHorizontal = 0
8 mMarginVertical = 0
9 mLabelQColor = "#FF0000"
10
11 INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
12 case = 2
13
14 def add_copyright(p, text, xOffset, yOffset):
15     p.translate( xOffset , yOffset )
16     text.drawContents(p)
17     p.setWorldTransform( p.worldTransform() )
18
19 def _on_render_complete(p):
20     deviceHeight = p.device().height() # Get paint device height on which this_
21     ↪painter is currently painting
22     deviceWidth = p.device().width() # Get paint device width on which this_
23     ↪painter is currently painting
24     # Create new container for structured rich text
25     text = QTextDocument()
26     font = QFont()
27     font.setFamily(mQFont)
28     font.setPointSize(int(mQFontsize))
29     text.setDefaultFont(font)
30     style = "<style type=\"text/css\"> p {color: " + mLabelQColor + "}</style>"
31     text.setHtml( style + "<p>" + mLabelQString + "</p>" )
32     # Text Size
33     size = text.size()
34
35     # RenderMillimeters
36     pixelsInchX = p.device().logicalDpiX()
37     pixelsInchY = p.device().logicalDpiY()
38     xOffset = pixelsInchX * INCHES_TO_MM * int(mMarginHorizontal)
39     yOffset = pixelsInchY * INCHES_TO_MM * int(mMarginVertical)
40
41     # Calculate positions
42     if case == 0:
43         # Top Left
44         add_copyright(p, text, xOffset, yOffset)
```

(continua na próxima página)

```

44     elif case == 1:
45         # Bottom Left
46         yOffset = deviceHeight - yOffset - size.height()
47         add_copyright(p, text, xOffset, yOffset)
48
49     elif case == 2:
50         # Top Right
51         xOffset = deviceWidth - xOffset - size.width()
52         add_copyright(p, text, xOffset, yOffset)
53
54     elif case == 3:
55         # Bottom Right
56         yOffset = deviceHeight - yOffset - size.height()
57         xOffset = deviceWidth - xOffset - size.width()
58         add_copyright(p, text, xOffset, yOffset)
59
60     elif case == 4:
61         # Top Center
62         xOffset = deviceWidth / 2
63         add_copyright(p, text, xOffset, yOffset)
64
65     else:
66         # Bottom Center
67         yOffset = deviceHeight - yOffset - size.height()
68         xOffset = deviceWidth / 2
69         add_copyright(p, text, xOffset, yOffset)
70
71     # Emitted when the canvas has rendered
72     iface.mapCanvas().renderComplete.connect(_on_render_complete)
73     # Repaint the canvas map
74     iface.mapCanvas().refresh()

```

21.11 Compositor

Get print layout by name

```

1 composerTitle = 'MyComposer' # Name of the composer
2
3 project = QgsProject.instance()
4 projectLayoutManager = project.layoutManager()
5 layout = projectLayoutManager.layoutByName(composerTitle)

```

21.12 Sources

- [QGIS Python \(PyQGIS\) API](#)
- [QGIS C++ API](#)
- [StackOverFlow QGIS questions](#)
- [Script de Klas Karlsson](#)