PyQGIS 3.34 developer cookbook

QGIS Project

20 de mayo de 2025

Índice general

1 Introducción **3**
- 1.1 Desarrollar scripts en la consola de Python . . . 4
- 1.2 Plugins Python . . . 4
  - 1.2.1 Complementos de procesamiento . . . 5
- 1.3 Ejecutar código Python cuando QGIS se inicia. . . . 5
  - 1.3.1 El fichero `startup.py` . . . 5
  - 1.3.2 La variable de entorno PYQGIS_STARTUP . . . 5
  - 1.3.3 El parámetro `--code` . . . 5
  - 1.3.4 Argumentos adicionales para Python . . . 6
- 1.4 Aplicaciones Python . . . 6
  - 1.4.1 Usando PyQGIS en scripts individuales . . . 6
  - 1.4.2 Usando PyQGIS en aplicaciones personalizadas . . . 7
  - 1.4.3 Ejecutar aplicaciones personalizadas . . . 8
- 1.5 Notas técnicas sobre PyQt y SIP . . . 8

2 Cargar proyectos **9**
- 2.1 Resolver rutas erróneas . . . 10
- 2.2 Uso de banderas para acelerar las cosas . . . 11

3 Cargar capas **13**
- 3.1 Capas Vectoriales . . . 13
- 3.2 Capas ráster . . . 16
- 3.3 Instancia QgsProject . . . 18

4 Accediendo a la Tabla de Contenidos (TOC) **19**
- 4.1 La clase QgsProject . . . 19
- 4.2 Clase QgsLayerTreeGroup . . . 20

5 Usar las capas ráster **23**
- 5.1 Detalles de la capa . . . 23
- 5.2 Renderizador . . . 24
  - 5.2.1 Rásters de una sola banda . . . 25
  - 5.2.2 Rásters multibanda . . . 26
- 5.3 Valores de consulta . . . 26
- 5.4 Edición de datos ráster . . . 26

6 Usar capas vectoriales **27**
- 6.1 Recuperando información sobre atributos . . . 28
- 6.2 Iterando sobre la capa vectorial . . . 28
- 6.3 Seleccionando objetos espaciales . . . 29
  - 6.3.1 Accediendo a atributos . . . 30

6.3.2 Iterando sobre rasgos seleccionados . . . 30
6.3.3 Iterando sobre un subconjunto de rasgos . . . 30
6.4 Modificación de capas vectoriales . . . 31
6.4.1 Añadir Entidades . . . 32
6.4.2 Borrar Entidades . . . 32
6.4.3 Modificar los objetos espaciales . . . 33
6.4.4 Modificación de capas vectoriales con un búfer de edición . . . 33
6.4.5 Agregando y Removiendo Campos . . . 34
6.5 Usar índice espacial . . . 35
6.6 La clase QgsVectorLayerUtils . . . 36
6.7 Creación de capas vectoriales . . . 36
6.7.1 Desde una instancia de `QgsVectorFileWriter` . . . 37
6.7.2 Directamente desde las funciones . . . 38
6.7.3 Desde una instancia de `QgsVectorLayer` . . . 39
6.8 Apariencia (Simbología) de capas vectoriales . . . 40
6.8.1 Representador de Símbolo Único . . . 41
6.8.2 Representador de símbolo categorizado . . . 42
6.8.3 Renderizador de símbolo graduado . . . 43
6.8.4 Trabajo con Símbolos . . . 44
6.8.5 Crear Renderizados personalizados . . . 47
6.9 Más Temas . . . 49

7 Manejo de Geometría **51**
7.1 Construcción de Geometría . . . 52
7.2 Acceso a Geometría . . . 52
7.3 Geometría predicados y Operaciones . . . 54

8 Soporte de Proyecciones **57**
8.1 Sistemas de coordenadas de referencia . . . 57
8.2 Transformación SRC . . . 59

9 Usando el Lienzo de Mapa **61**
9.1 Lienzo de mapa insertado . . . 62
9.2 Bandas elásticas y marcadores de vértices . . . 63
9.3 Utilizar las herramientas del mapa con el lienzo . . . 64
9.3.1 Selecciona una entidad usando QgsMapToolIdentifyFeature . . . 65
9.3.2 Añadir temas al menú contextual del lienzo del mapa . . . 65
9.4 Escribir herramientas de mapa personalizados . . . 66
9.5 Escribir elementos de lienzo de mapa personalizado . . . 67

10 Representación del Mapa e Impresión **69**
10.1 Representación Simple . . . 70
10.2 Representando capas con diferente SRC . . . 70
10.3 Salida usando diseño de impresión . . . 71
10.3.1 Comprobación de la validez del diseño . . . 72
10.3.2 Exportando la composición . . . 73
10.3.3 Exportar un atlas de diseño . . . 74

11 Expresiones, Filtros y Calculando Valores **75**
11.1 Análisis de expresiones . . . 76
11.2 Evaluar expresiones . . . 76
11.2.1 Expresiones Basicas . . . 77
11.2.2 Expresiones con caracteristicas . . . 77
11.2.3 Flitrando una capa con expresiones . . . 78
11.3 Manejando errores de expresión . . . 79

12 Configuración de lectura y almacenamiento **81**

13 Comunicarse con el usuario **85**

13.1 Mostrando mensajes. La clase QgsMessageBar . . . 85
13.2 Mostrando el progreso . . . 88
13.3 Registro . . . 89
13.3.1 QgsMessageLog . . . 89
13.3.2 El python construído en el módulo de registro . . . 90

14 Infraestructura de autenticación **91**
14.1 Introducción . . . 92
14.2 Glosario . . . 92
14.3 QgsAuthManager el punto de entrada . . . 93
14.3.1 Inicie el administrador y configure la contraseña maestra . . . 93
14.3.2 Complete authdb con una nueva entrada de configuración de autenticación . . . 93
14.3.3 Borrar una entrada de authdb . . . 95
14.3.4 Deje la expansión authcfg a QgsAuthManager . . . 95
14.4 Adpatar complementos para usar Infraestructura de Autenticación . . . 96
14.5 Autenticación IGUs . . . 96
14.5.1 IGU para seleccionar credenciales . . . 96
14.5.2 IGU Editor Autenticación . . . 97
14.5.3 IGU de Editor de Autoridades . . . 98

15 Tareas - haciendo trabajo duro en segundo plano **99**
15.1 Introducción . . . 99
15.2 Ejemplos . . . 101
15.2.1 Extendiendo QgsTask . . . 101
15.2.2 Tarea desde función . . . 103
15.2.3 Tarea de un algoritmo de procesamiento . . . 104

16 Desarrollando Plugins Python **107**
16.1 Estructurar Complementos de Python . . . 107
16.1.1 Cómo empezar . . . 107
16.1.2 Escribir el código del complemento . . . 108
16.1.3 Documentación de complementos . . . 113
16.1.4 Traducción de complementos . . . 113
16.1.5 Compartir su complemento . . . 115
16.1.6 Consejos y Trucos . . . 115
16.2 Fragmentos de código . . . 116
16.2.1 Cómo llamar a un método por un atajo de teclado . . . 117
16.2.2 Cómo reutilizar los iconos de QGIS . . . 117
16.2.3 Interfaz para complemento en el cuadro de diálogo de opciones . . . 118
16.2.4 Incrustar widgets personalizados para capas en el árbol de capas . . . 119
16.3 Configuración IDE para escribir y depurar complementos . . . 120
16.3.1 Complementos útiles para escribir complementos de Python . . . 120
16.3.2 Una nota sobre la configuración de su IDE en Linux y Windows . . . 120
16.3.3 Depuración con Pyscripter IDE (Windows) . . . 121
16.3.4 Depurar con Eclipse y PyDev . . . 121
16.3.5 Depurar con PyCharm en Ubuntu con un QGIS compilado . . . 125
16.3.6 Depurar usando PDB . . . 127
16.4 Lanzamiento de su complemento . . . 127
16.4.1 Metadatos y nombres . . . 127
16.4.2 Código y ayuda . . . 128
16.4.3 Repositorio oficial de complementos de Python . . . 128

17 Escribir nuevos complementos de procesamiento **131**
17.1 Creando desde cero . . . 131
17.2 Actuializar un complemento . . . 131

18 Utilizar complemento Capas **135**
18.1 Subclassing QgsPluginLayer . . . 135

19 Biblioteca de análisis de redes **137**
19.1 Información general . . . 137
19.2 Contruir un gráfico . . . 138
19.3 Análisis gráfico . . . 140
19.3.1 Encontrar la ruta más corta . . . 142
19.3.2 Áreas de disponibilidad . . . 144

20 Servidor QGIS y Python **147**
20.1 Introducción . . . 147
20.2 Conceptos básicos de la API del servidor . . . 148
20.3 Independiente o incrustado . . . 148
20.4 Complementos del servidor . . . 149
20.4.1 Complementos de filtro de servidor . . . 149
20.4.2 Servicios personalizados . . . 157
20.4.3 APIs personalizadas . . . 158

21 hoja de referencia para PyQGIS **161**
21.1 Interfaz de Usuario . . . 161
21.2 Configuración . . . 162
21.3 Barras de herramientas . . . 162
21.4 Menús . . . 162
21.5 Lienzo . . . 163
21.6 Capas . . . 163
21.7 Tabla de contenidos . . . 167
21.8 TOC avanzado . . . 167
21.9 Algoritmos de procesamiento . . . 170
21.10 Decoradores . . . 171
21.11 Compositor . . . 172
21.12 Fuentes . . . 172

CAPÍTULO 1

Introducción

Este documento pretende ser tanto un tutorial y una guía de referencia. Aunque no enumera todos los casos de uso posibles, debe proporcionar una buena visión general de la funcionalidad principal.

Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre GNU, Versión 1.3 o cualquier versión posterior publicada por la Free Software Foundation; sin secciones invariantes, sin textos de portada y sin textos de contraportada.

Se incluye una copia de la licencia en la sección gnu_fdl.

Esta licencia también se aplica a todos los fragmentos de código de este documento.

El soporte de Python se introdujo por primera vez en QGIS 0.9. Hay varias maneras de utilizar Python en QGIS Desktop (cubierto en las siguientes secciones):

- Emita comandos en la consola de Python dentro de QGIS
- Crear y usar plugins
- Ejecute automáticamente código Python cuando se inicie QGIS
- Crear algoritmos de procesamiento
- Crear funciones para expresiones en QGIS
- Crear aplicaciones personalizadas basadas en la API de QGIS

Los enlaces de Python también están disponibles para QGIS Server, incluidos los plugins de Python (vea *Servidor QGIS y Python*) y los enlaces de Python que se pueden usar para incrustar QGIS Server en una aplicación de Python.

Hay una referencia API completa de QGIS C++ que documenta las clases de las librerías de QGIS. The Pythonic QGIS API (pyqgis) es casi idéntica a la API de C++.

Otro buen recurso para aprender a realizar tareas comunes es descargar complementos existentes desde el repositorio de complementos y examine su código.

1.1 Desarrollar scripts en la consola de Python

QGIS provee una consola Python integrada para scripting. Se puede abrir desde el menu: *Complementos* ► *Consola de Python*

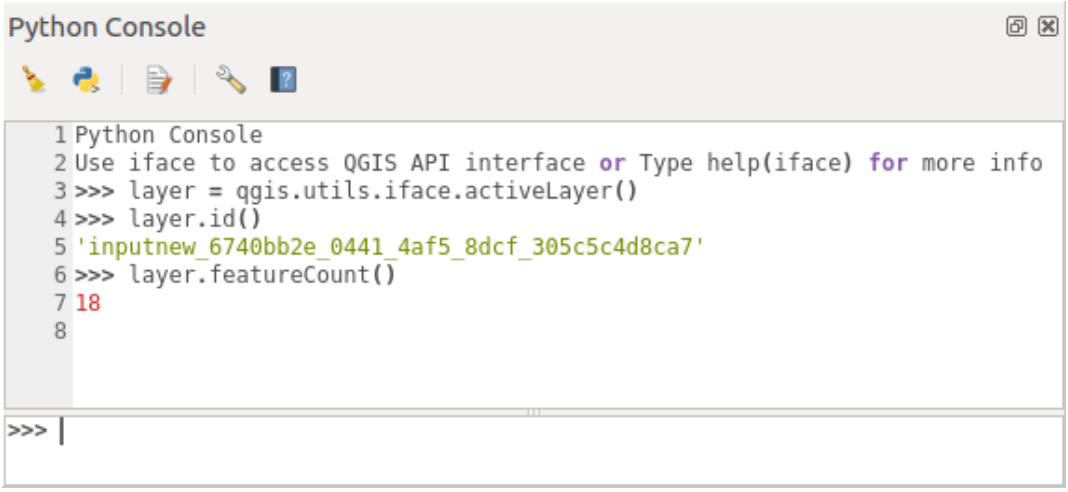


Figura 1.1: Consola Python de QGIS

La captura de pantalla anterior ilustra cómo obtener la capa seleccionada actualmente en la lista de capas, mostrar su ID y opcionalmente, si se trata de una capa vectorial, mostrar el recuento de entidades. Para la interacción con el entorno QGIS, hay una variable `iface` que es una instancia de la clase: *QgisInterface <qgis.gui.QgisInterface>*. Esta interfaz permite el acceso al lienzo del mapa, menús, barras de herramientas y otras partes de la aplicación QGIS.

Para mayor comodidad del usuario, las siguientes instrucciones se ejecutan cuando se inicia la consola (en el futuro será posible establecer más comandos iniciales)

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

Para aquellos que utilizan la consola a menudo, puede ser útil establecer un acceso directo para activar la consola (dentro de *Configuración* ► *Atajos de teclado…*)

1.2 Plugins Python

La funcionalidad de QGIS se puede ampliar utilizando complementos. Los complementos se pueden escribir en Python. La principal ventaja sobre los complementos de C++ es la simplicidad de la distribución (sin compilación para cada plataforma) y el desarrollo más fácil.

Muchos complementos que cubren diversas funciones se han escrito desde la introducción del soporte de Python. El instalador de complemento permite a los usuarios buscar, actualizar y eliminar fácilmente complementos Python. Consulte la página Complementos de Python para más información sobre complementos y desarrollo de complementos.

Crear plugins con Python es simple, vea *Desarrollando Plugins Python* para instrucciones detalladas.

Nota: Los plugins de Python están también disponibles para QGIS Server. Vea *Servidor QGIS y Python* para más detalles.

1.2.1 Complementos de procesamiento

Los complementos de procesamiento se pueden utilizar para procesar datos. Son más fáciles de desarrollar, más específicos y más livianos que los complementos de Python. *Escribir nuevos complementos de procesamiento* explica cuándo es adecuado el uso de algoritmos de Processing y cómo desarrollarlos.

1.3 Ejecutar código Python cuando QGIS se inicia.

Existen diferentes métodos para ejecutar código Python cada vez que se inicia QGIS.

1. Crear un script startup.py
2. Configurar la variable de entorno `PYQGIS_STARTUP` a un fichero Python existente
3. Especificación de un script de inicio mediante el parámetro `--code init_qgis.py`.

1.3.1 El fichero `startup.py`

Cada vez que se inicia QGIS, se busca en el directorio de Python del usuario y en una lista de rutas del sistema un fichero llamado `startup.py`. Si ese archivo existe, es ejecutado por el intérprete de Python incrustado.

La ruta en el directorio personal del usuario suele encontrarse en:

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

Las rutas predeterminada del sistema dependen del sistema operativo. Para encontrar las rutas que funcionan para usted, abra la consola de Python y ejecute `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` para ver la lista de directorios por defecto.

El script `startup.py` se ejecuta inmediatamente después de inicializar python en QGIS, al inicio de la aplicación.

1.3.2 La variable de entorno PYQGIS_STARTUP

Al configurar la variable de entorno `PYQGIS_STARTUP` con la ruta de un fichero Python existente, puede ejecutar código Python justo antes de que la inicialización de QGIS se haya completado.

Este código se ejecutará antes de que se complete la inicialización de QGIS. Este método es muy útil para la limpieza de sys.path, que puede tener rutas no deseables, o para aislar/cargar el entorno inicial sin necesidad de un entorno virtual, por ejemplo, homebrew o MacPorts se instala en Mac.

1.3.3 El parámetro `--code`

Puede proporcionar código personalizado para ejecutar como parámetro de inicio a QGIS. Para ello, cree un archivo python, por ejemplo `qgis_init.py`, para ejecutar e iniciar QGIS desde la línea de comandos usando `qgis --code qgis_init.py`.

El código proporcionado a través de `--code` se ejecuta al final de la fase de inicialización de QGIS, después de que se hayan cargado los componentes de la aplicación.

1.3.4 Argumentos adicionales para Python

Para proporcionar argumentos adicionales para tu script `--code` o para otro código python que se ejecute, puedes usar el argumento `--py-args`. Cualquier argumento que venga después de `--py-args` y antes de `--` arg (si está presente) será pasado a Python pero ignorado por la propia aplicación QGIS.

En el siguiente ejemplo, `myfile.tif` estará disponible a través de `sys.argv` en Python pero no será cargado por QGIS. Mientras que `otherfile.tif` será cargado por QGIS pero no está presente en `sys.argv`.

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

Si quiere acceder a todos los parámetros de la línea de comandos desde Python, puedes utilizar `QCoreApplication.arguments()`.

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

1.4 Aplicaciones Python

A menudo es útil crear scripts para automatizar procesos. Con PyQGIS, esto es perfectamente posible — importe el módulo qgis.core, Inicialícelo y estará listo para el procesamiento.

O tal vez desee crear una aplicación interactiva que utilice la funcionalidad GIS — realizar mediciones, exportar un mapa como PDF, … El módulo qgis.gui proporciona varios componentes GUI, en particular el widget de lienzo del mapa que se puede incorporar a la aplicación con soporte para hacer zoom, paneo y/o cualquier otra herramienta de mapa personalizada.

Las aplicaciones personalizadas de PyQGIS o los scripts independientes deben configurarse para localizar los recursos QGIS, como la información de proyección y los proveedores para leer capas vectoriales y ráster. Los recursos de QGIS se inician añadiendo unas pocas líneas al principio de su aplicación o script. El código para inicializar QGIS para aplicaciones personalizadas y scripts independientes es similar. A continuación se proporcionan ejemplos de cada uno de ellos.

Nota: *No* utilice `qgis.py` como nombre para su script. Python no podrá importar los enlaces, ya que el nombre del script será su sombra.

1.4.1 Usando PyQGIS en scripts individuales

Para iniciar un script independiente, inicialice los recursos QGIS al principio del script:

```
from qgis.core import *

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

# Create a reference to the QgsApplication.  Setting the
# second argument to False disables the GUI.
qgs = QgsApplication([], False)

# Load providers
qgs.initQgis()

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

# Finally, exitQgis() is called to remove the
# provider and layer registries from memory
qgs.exitQgis()
```

Primero importa el módulo qgis.core y configura el prefijo de ruta. El prefijo de ruta es la ubicación donde está instalado QGIS en su sistema. Se configura en el script llamando al método `setPrefixPath()`. El segundo argumento de `setPrefixPath()` es establecido a `True`, especificando que se utilizarán las rutas predeterminadas.

La ruta de instalación de QGIS varía según la plataforma; la forma más fácil de encontrarlo para su sistema es usar *Desarrollar scripts en la consola de Python* desde dentro de QGIS y ver el resultado de ejecutar:

```
QgsApplication.prefixPath()
```

Una vez configurada la ruta del prefijo, guardamos una referencia a `QgsApplication` en la variable `qgs`. El segundo argumento está establecido como `False`, especificando que no planeamos usar la GUI ya que estamos escribiendo un script independiente. Con `QgsApplication` configurado, cargamos los proveedores de datos de QGIS y el registro de capas llamando al método `initQgis()`.

```
qgs.initQgis()
```

Con QGIS inicializado, estamos listos para escribir el resto del script. Finalmente, terminamos llamando a `exitQgis()` para eliminar los proveedores de datos y el registro de capas de la memoria.

```
qgs.exitQgis()
```

1.4.2 Usando PyQGIS en aplicaciones personalizadas

La única diferencia entre *Usando PyQGIS en scripts individuales* y una aplicación PyQGIS personalizada es el segundo argumento al crear una instancia del `QgsApplication`. Pase `True` en lugar de `False` para indicar que planeamos utilizar una IGU.

```
from qgis.core import *

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

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

qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

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

# Finally, exitQgis() is called to remove the
# provider and layer registries from memory
qgs.exitQgis()
```

Ahora puede trabajar con la API de QGIS - carga de capas y realizar algún procesamiento o encender una GUI con un lienzo de mapa. Las posibilidades son infinitas :-)

1.4.3 Ejecutar aplicaciones personalizadas

Necesita indicar a su sistema dónde buscar las bibliotecas QGIS y módulos de Python apropiados si no están en una ubicación conocida - de lo contrario Python se quejará:

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

Esto se puede solucionar estableciendo la variable de entorno `PYTHONPATH` . En los siguientes comandos, `<qgispath>` deberá ser reemplazado con su ruta de instalación de QGIS actual:

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

Ahora, se conoce la ruta de acceso a los módulos PyQGIS, pero dependen de las bibliotecas `qgis_core` y `qgis_gui` (los módulos de Python solo sirven como contenedores). La ruta de acceso a estas bibliotecas puede ser desconocida para el sistema operativo, y luego obtendrá un error de importación de nuevo (el mensaje puede variar dependiendo del sistema):

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

Para solucionar, agregar los directorios donde residen las bibliotecas QGIS a la ruta de búsqueda del enlazador dinámico:

- en Linux: **`export LD_LIBRARY_PATH=/<qgispath>/lib`**
- en Windows: **`set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%`** donde `<qgisrelease>` debe ser reemplazado por el tipo de enlace apuntado (por ejemplo: `qgis-ltr`, `qgis`, `qgis-dev`)

Estos comandos se pueden poner en un script de arranque que se encargará del inicio. Al implementar aplicaciones personalizadas con PyQGIS, normalmente hay dos posibilidades:

- requiere que el usuario instale QGIS antes de instalar la aplicación. El instalador de la aplicación debe buscar ubicaciones predeterminadas de las bibliotecas QGIS y permitir al usuario establecer la ruta si no se encuentra. Este enfoque tiene la ventaja de ser más sencillo, sin embargo, requiere que el usuario haga más pasos.
- paquete QGIS junto con su aplicación. Lanzar la aplicación puede ser más difícil y el paquete será más grande, pero el usuario se salvará de la carga de descargar e instalar piezas adicionales de software.

Los dos modelos de implementación pueden ser mixtos. Puede proporcionar aplicaciones independientes en Windows y macOS, pero para Linux dejar la instalación de SIG en manos del usuario y su administrador de paquetes.

1.5 Notas técnicas sobre PyQt y SIP

Se ha decidido por Python, ya que es uno de los lenguajes más favoritos para el scripting. Los enlaces PyQGIS en QGIS 3 dependen de SIP y PyQt5. La razón para usar SIP en lugar del ampliamente utilizado SWIG es que el código QGIS depende de las bibliotecas Qt. Los enlaces de Python para Qt (PyQt) se realizan utilizando SIP y esto permite la integración perfecta de PyQGIS con PyQt.

CAPÍTULO 2

Cargar proyectos

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

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

Algunas veces se necesita cargar un proyecto existente desde un complemento o (más a menudo) al desarrollar una aplicación autónoma QGIS Python (vea : *Aplicaciones Python*).

Para cargar un proyecto en la aplicación QGIS actual, debe crear una instancia de la clase `QgsProject`. Esta es una clase singleton, por lo tanto se debe usar el método `instance()` para realizarlo. Puede llamar su método `read()` y pasar la ruta para que el proyecto sea cargado:

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

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

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

Si necesita hacer modificaciones a su proyecto (por ejemplo añadir o remover algunas capas) y guardar los cambios realizados, puede llamar el método `write()` de su instancia de proyecto. El método `write()` también acepta una

ruta opcional para salvar el proyecto en una nueva localización:

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

Las funciones `read()` y `write()` retornan un valor booleano que puede utilizar para verificar si la operación fue exitosa.

Nota: Si está desarrollando una aplicación QGIS autónoma, para poder mantener la sincronización entre el proyecto cargado y el lienzo, debe instanciar una :class:"QgsLayerTreeMapCanvasBridge <qgis.gui.QgsLayerTreeMapCanvasBridge>" al igual que en el ejemplo:

```
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 Resolver rutas erróneas

Puede suceder que las capas cargadas en el proyecto se muevan a otra ubicación. Cuando el proyecto se vuelve a cargar, todas las rutas de las capas se rompen. La clase `QgsPathResolver` le ayuda a reescribir la ruta de las capas dentro del proyecto.

Su método `setPathPreprocessor()` permite configurar una función de preprocesador de ruta personalizada para manipular rutas y fuentes de datos antes de resolverlas en referencias de archivos o fuentes de capas.

La función del procesador debe aceptar un único argumento de cadena (que represente la ruta del archivo original o la fuente de datos) y devolver una versión procesada de esta ruta. La función de preprocesador de ruta se llama **antes** de cualquier controlador de capa defectuoso. Si se configuran varios preprocesadores, se llamarán en secuencia según el orden en que se configuraron originalmente.

Algunos casos de uso:

1. reemplazar ruta desactualizada:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

2. reemplace una dirección host de base de datos con una nueva:

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. reemplazar credenciales de base de datos almacenada con unas nuevas:

```
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)
```

Del mismo modo, un método `setPathWriter()` está disponible para una función de escritor de ruta.

Un ejemplo para reemplazar la ruta con una variable:

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

QgsPathResolver.setPathWriter(my_processor)
```

Ambos métodos devuelven un `id` que se puede usar para eliminar el preprocesador o escritor que agregaron. Consulte `removePathPreprocessor()` y `removePathWriter()`.

2.2 Uso de banderas para acelerar las cosas

En algunos casos en los que puede que no necesite utilizar un proyecto completamente funcional, pero sólo desea acceder a él por una razón específica, las banderas pueden ser útiles. Una lista completa de banderas está disponible en `ProjectReadFlag`. Se pueden añadir varias banderas juntas.

Por ejemplo, si no nos importan las capas y los datos reales y simplemente queremos acceder a un proyecto (por ejemplo, para el diseño o la configuración de la vista 3D), podemos utilizar la bandera `DontResolveLayers` para omitir el paso de validación de datos y evitar que aparezca el diálogo de capa incorrecta. Se puede hacer lo siguiente:

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

Para añadir más banderas se debe utilizar el operador Bitwise OR de python (`|`).

CAPÍTULO 3

Cargar capas

Consejo: Los fragmentos de código en esta página necesitan las siguientes importaciones:

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

Vamos a abrir algunas capas con datos. QGIS reconoce capas vectoriales y ráster. Además, están disponibles tipos de capas personalizadas, pero no se va a discutir de ellas aquí.

3.1 Capas Vectoriales

Para crear y agregar una instancia de capa vectorial al proyecto, especifique el identificador de origen de datos de la capa, el nombre de la capa y el nombre del proveedor:

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

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

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

El identificador de la fuente de datos es una cadena y se especifica a cada proveedor de datos vectoriales. El nombre de la capa se utiliza en el widget de la lista de capa. Es importante validar si la capa se ha cargado satisfactoriamente. Si no fue así, se devuelve una instancia de capa no válida.

Para una capa vectorial creada mediante un geopackage:

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

La forma más rápida para abrir y visualizar una capa vectorial en QGIS es usar el método `addVectorLayer()` perteneciente a `QgisInterface`:

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

Esto crea una nueva capa y la añade al actual proyecto QGIS (haciéndola aparecer en el listado de capas) en un solo paso. La función retorna la instancia de capa o *None*` si es que no puede cargarla.

La siguiente lista muestra cómo acceder a varias fuentes de datos utilizando los proveedores de datos vectoriales:

- Biblioteca GDAL (Shapefile y muchos otros formatos de archivo) — la fuente de datos es la ruta al archivo:
  - Para Shapefile:

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

  - Para dxf (tenga en cuenta las opciones internas en la fuente de datos uri):

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

- Base de datos PostGIS - la fuente de datos es una cadena de texto con toda la información necesaria para crear una conexión con la base de datos PostgreSQL.

  La clase `QgsDataSourceUri` puede generar esta cadena de texto para usted. Tenga en cuenta que QGIS debe compilarse con el soporte de Postgres, o de lo contrario, este proveedor no estará disponible:

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

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

  Nota: El argumento `False` pasado a `uri.uri(False)` previene la expansión de los parámetros de configuración de la autenticación. En caso de que no esté utilizando ninguna configuración para autenticación, este argumento no hará ninguna diferencia.

- CSV u otros archivos de texto delimitados — para abrir un archivo con un punto y coma como delimitador, con el campo «x» para la coordenada X y el campo «y» para la coordenada Y, usaría algo como esto:

```
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: La cadena de proveedor está estructurada como una dirección URL, por lo que la ruta de acceso debe ir precedida de `file:///`. También permite geometrías en formato WKT (texto bien conocido) como alternativa a los campos “”x”” y “”y””, y permite especificar el sistema de referencia de coordenadas. Por ejemplo:

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

- Los archivos GPX — el proveedor de datos «gpx» lee los caminos, rutas y puntos de interés desde archivos GPX. Para abrir un archivo, el tipo (caminos/ruta/punto de interés) se debe especificar como parte de la url:

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

- La base de datos SpatiaLite — De forma similar a las bases de datos PostGIS, :class:”QgsDataSourceUri <qgis.core.QgsDataSourceUri>” puede ser utilizado para la generación de identificador de origen de datos:

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

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

- Geometrías basadas en MySQL WKB, a través de GDAL — la fuente de datos es la cadena de conexión a la tabla:

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

- Conexión WFS: La conexión se define con un URL y usando el proveedor `` WFS “”:

```
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")
```

La uri se puede crear utilizando la librería estándar `urllib`:

```
import urllib

params = {
    'service': 'WFS',
    'version': '2.0.0',
    'request': 'GetFeature',
    'typename': 'ms:cities',
    'srsname': "EPSG:4326"
}
uri2 = 'https://demo.mapserver.org/cgi-bin/wfs?' + urllib.parse.unquote(urllib.
↪parse.urlencode(params))
```

Nota: Puede cambiar la fuente de datos de una capa existente llamando a `setDataSource()` en una instancia de `QgsVectorLayer`, como se muestra a continuación ejemplo:

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

del(vlayer)
```

3.2 Capas ráster

Para acceder a un archivo raster, se utiliza la librería GDAL. Esta soporta un amplio rango de formatos de archivo. En caso de que tenga problemas al abrir algún archivo, compruebe si es que su GDAL tiene soporte para el formato en particular (no todos los formatos están disponibles de forma predeterminada). Para cargar un raster desde un archivo, especifique el nombre del archivo y su nombre de visualización:

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

Para cargar una capa raster desde un geopackage:

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

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

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

De manera similar a las capas vectoriales, las capas raster pueden ser cargadas utilizando la función addRasterLayer de un objeto perteneciente a `QgisInterface`

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

Esto crea una nueva capa y la añade al proyecto actual (haciendo que aparezca en la lista) en un solo paso.

Para cargar un ráster PostGIS:

Los rásteres PostGIS, similares a los vectores PostGIS, se pueden agregar a un proyecto mediante una cadena URI. Es eficaz mantener un diccionario de cadenas reutilizable para los parámetros de conexión de la base de datos. Esto facilita la edición del diccionario para la conexión correspondiente. Luego, el diccionario se codifica en un URI utilizando el objeto de metadatos del proveedor “postgresraster”. Después de eso, el ráster se puede agregar al proyecto.

```
uri_config = {
    # database parameters
    'dbname':'gis_db',       # The PostgreSQL database to connect to.
    'host':'localhost',      # The host IP address or localhost.
    'port':'5432',           # The port to connect on.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    'sslmode':QgsDataSourceUri.SslDisable, # SslAllow, SslPrefer, SslRequire, SslVerifyCa, SslVerifyFull
    # user and password are not needed if stored in the authcfg or service
    'authcfg':'QconfigId',  # The QGIS athentication database ID holding connection details.
    'service': None,          # The PostgreSQL service to be used for connection to the database.
    'username':None,        # The PostgreSQL user name.
    'password':None,        # The PostgreSQL password for the user.
    # table and raster column details
    'schema':'public',      # The database schema that the table is located in.
    'table':'my_rasters',   # The database table to be loaded.
    'geometrycolumn':'rast',# raster column in PostGIS table
    'sql':None,             # An SQL WHERE clause. It should be placed at the end of the string.
    'key':None,             # A key column from the table.
    'srid':None,            # A string designating the SRID of the coordinate reference system.
    'estimatedmetadata':'False', # A boolean value telling if the metadata is estimated.
    'type':None,            # A WKT string designating the WKB Type.
    'selectatid':None,      # Set to True to disable selection by feature ID.
    'options':None,         # other PostgreSQL connection options not in this list.
    'enableTime': None,
    'temporalDefaultTime': None,
    'temporalFieldIndex': None,
    'mode':'2',             # GDAL 'mode' parameter, 2 unions raster tiles, 1 adds tiles separately (may require user input)
}
# remove any NULL parameters
uri_config = {key:val for key, val in uri_config.items() if val is not None}
# get the metadata for the raster provider and configure the URI
md = QgsProviderRegistry.instance().providerMetadata('postgresraster')
uri = QgsDataSourceUri(md.encodeUri(uri_config))

# the raster can then be loaded into the project
rlayer = iface.addRasterLayer(uri.uri(False), "raster layer name", "postgresraster")
```

Las capas ráster también se pueden crear desde el servicio WCS:

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

Aquí está una descripción de los parámetros que el WCS URI puede contener:

El WCS URI se compone de pares **calve=valor** separadas por `&`. Es el mismo formato que la cadena de consulta en la URL, codificada de la misma manera. `QgsDataSourceUri` debe utilizarse para construir el URI para garantizar que los caracteres especiales se codifican correctamente.

- **url** (requerido) : URL del servidor WCS. No utilice la VERSION en el URL, porque cada versión del WCS está usando nombre de parámetro diferente para la versión de **GetCapabilities** vea la versión del parámetro.
- **identifier** (requerido) : Nombre de la Cobertura
- **time** (opcional) : posición de tiempo o período de tiempo (beginPosition/endPosition[/timeResolution])
- **format** (opcional) : Nombre de formato admitido. El valor predeterminado es el primer formato compatible con el nombre en tif o el primer formato compatible.
- **crs** (opcional): CRS en el formato AUTORIDAD:IDENTIFICADOR, p. ej. EPSG:4326. El valor predeterminado es EPSG:4326, si es que es compatible, o si no el primer CRS compatible.

- **nombre de usuario** (opcional): Nombre de usuario para la autenticación básica.
- **contraseña** (opcional): Contraseña para la autenticación básica.
- **IgnoreGetMapUrl** (opcional, hack): si se especifica (establecido en 1), ignore la dirección URL de GetCoverage anunciada por GetCapabilities. Puede ser necesario si un servidor no está configurado correctamente.
- **InvertAxisOrientation** (opcional, hack): si se especifica (establecido en 1), cambie el eje en la solicitud GetCoverage. Puede ser necesario para un CRS geográfico si un servidor está utilizando un orden de eje incorrecto.
- **IgnoreAxisOrientation** (opcional, hack): Si se especifica (establecido en 1), no invierta la orientación del eje de acuerdo con el estándar WCS para un CRS geográfico.
- **cache** (opcional): control de carga de caché, como se describe en QNetworkRequest::CacheLoadControl, pero la solicitud se reenvía como PreferCache si falló con AlwaysCache. Valores permitidos: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. El valor predeterminado es AlwaysCache.

Como alternativa se puede cargar una capa ráster desde un servidor WMS. Sin embargo actualmente no es posible acceder a las respuestas de GetCapabilities desde el API — se debe saber que capas desea:

```
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 Instancia QgsProject

Si desea utilizar las capas abiertas para la representación, no olvide agregarlas a la instancia de `QgsProject`. La instancia `QgsProject` toma la posesión de las capas y más adelante, se puede acceder desde cualquier parte de la aplicación mediante su identificador único . Cuando la capa se elimina del proyecto, también se elimina. Las capas pueden ser eliminadas por el usuario en la interfaz QGIS, o a través de Python usando el método `removeMapLayer()`.

Añadir una capa al proyecto actual, se puede realizar, utilizando el método `addMapLayer()` :

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

Para agregar una capa en una posición absoluta:

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

Si quiere remover una capa utilice el método `removeMapLayer()` :

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

En el código anterior, el identificador de la capa es pasado (puede obtenerlo llamando el método `id()` que pertenece a la capa), pero también puede hacerlo pasando el objeto capa en si mismo.

Para una lista de capas cargadas y sus identificadores, use el método `mapLayers()` :

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

CAPÍTULO 4

Accediendo a la Tabla de Contenidos (TOC)

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

Puede usar diferentes clases para acceder a todas las capas cargadas en la TOC y usarlas para obtener información:

- `QgsProject`
- `QgsLayerTreeGroup`

4.1 La clase QgsProject

Puede usar la clase `QgsProject` para recuperar información sobre la TOC y todas las capas cargadas.

Debe crear una `instance` de `QgsProject` y usar sus métodos para obtener las capas cargadas.

El método principal es `mapLayers()`. Devolverá un diccionario con las capas cargadas:

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

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

Las `claves` del diccionario son las ids únicas de capa mientras que los `valores` son los objetos relacionados.

Ahora es sencillo obtener cualquier otra información sobre las capas:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

print(layers_list)
```

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

Puede además consultar la TOC usando el nombre de la capa:

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

Nota: Se devuelve una lista con todas las capas coincidentes, por lo que indexamos con `[0]` para obtener la primera capa con este nombre.

4.2 Clase QgsLayerTreeGroup

La capa árbol es una estructura clásica de árbol construida con nodos. Hay actualmente dos tipos de nodos: nodos de grupo (`QgsLayerTreeGroup`) y nodos de capa (`QgsLayerTreeLayer`).

Nota: para mas información puede leer este blog puesto por Martin Dobias: Part 1 Part 2 Part 3

Se puede acceder fácilmente al árbol de capas del proyecto con el método `layerTreeRoot()` de la clase `QgsProject`:

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

`root` es un nodo de grupo y tiene *hijos*:

```
root.children()
```

Se devuelve una lista de hijos directos. Se debe acceder a los hijos del subgrupo desde su propio padre directo.

Podemos recuperar uno de los hijos:

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

```
<QgsLayerTreeLayer: countries>
```

Las capas también se pueden recuperar usando su (única) `id`:

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

Y los grupos también se pueden buscar usando sus nombres:

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

`QgsLayerTreeGroup` tiene muchos otros métodos útiles que se pueden utilizar para obtener más información sobre el TOC:

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

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

Ahora agreguemos algunas capas al árbol de capas del proyecto. Hay dos formas de hacerlo:

1. **Adición explícita** usando las funciones `addLayer()` o `insertLayer()`:

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

2. **Adición implícita**: dado que el árbol de capas del proyecto está conectado al registro de capas, basta con agregar una capa al registro de capas del mapa:

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

Puede conmutar entre `QgsVectorLayer` y `QgsLayerTreeLayer` fácilmente:

```
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)>
```

Los grupos pueden ser añadidos con el método `addGroup()`. En el siguiente ejemplo, el primero agregará un grupo al final de la tabla de contenido, mientras que para el segundo puede agregar otro grupo dentro de uno 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 nodos y grupos hay muchos métodos útiles.

Mover un nodo existente se hace en tres pasos:

1. Clonando el nodo existente
2. moviendo el nodo clonado a la posición deseada
3. borrando el nodo original

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

Es un poco mas *complicado* mover una capa por la leyenda:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

o moverla a un grupo existente:

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

Algunos otros métodos que se pueden utilizar para modificar los grupos y capas:

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

CAPÍTULO 5

Usar las capas ráster

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

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

5.1 Detalles de la capa

Una capa ráster está compuesta por una o más bandas ráster — denominadas como raster monobanda o multibanda. Una banda representa una matriz de valores. Una imagen a color (p. ej. una fotografía aérea) es un ráster que está constituido por bandas roja, azul y verde. Los rásteres de banda única, representan generalmente variables continuas (p. ej. elevación) o variables discretas (p. ej. uso del suelo). En algunos casos, una capa ráster viene con una paleta y los valores ráster se refieren a los colores almacenados en la paleta.

El código a continuación, asume que `rlayer` es un objeto de `QgsRasterLayer` .

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

```
919 619
```

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

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

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

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

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

```
RasterLayerType.GrayOrUndefined
```

```
 # 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

Cuando una capa ráster es cargada, recibe en base a su tipo, el valor del renderizador de forma predeterminada. Esto puede ser modificado tanto en las propiedades de capa o mediante programación.

Para consultar el actual renderizador:

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

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

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

```
singlebandgray
```

Para establecer un renderizador, use el método `setRenderer()` de `QgsRasterLayer`. Hay una serie de clases de renderizado (derivadas de `QgsRasterRenderer`):

- `QgsHillshadeRenderer`
- `QgsMultiBandColorRenderer`

- QgsPalettedRasterRenderer
- QgsRasterContourRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

Las capas ráster de banda única pueden ser dibujadas tanto en colores grises (valor menor = negro, valor alto = blanco) o con un algoritmo pseudocolor que asigna colores a los valores. Rásters de banda única con una paleta puedes ser dibujados usando la paleta. Las capas multibanda generalmente se dibujan asignando las bandas a colores RGB. Otra posibilidad es usar una sola banda para dibujar.

5.2.1 Rásters de una sola banda

Supongamos que queremos renderizar una capa ráster de una sola banda con colores que van del verde al amarillo (correspondiente a los valores de píxel de 0 a 255). En la primera etapa prepararemos un objeto QgsRasterShader y configuraremos su función shader:

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

El sombreador asigna los colores según lo especificado por su mapa de colores. El mapa de colores es proveído como una lista de valores de pixeles con colores asociados. Hay tres modos de interpolación:

- lineal (`Interpolated`): el color es linealmente interpolado desde las entradas en el mapa de colores, que están por encima y por debajo de el valor de pixel.
- discreto (`Discrete`): el color es tomado desde la entrada más cercana con igual o mayor valor en el mapa de colores.
- exacto (`Exact`): el color no es interpolado, solo los pixeles con un valor igual a las entradas del mapa de colores serán dibujados.

En el segundo paso asociaremos el sombreador con una capa ráster:

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

El número `1` en el código anterior es el número de la banda (bandas ráster son indexadas de uno).

Finálmente tenemos que usar el método `triggerRepaint()` para ver los resultados:

```
rlayer.triggerRepaint()
```

5.2.2 Rásters multibanda

De forma predeterminada, QGIS asigna las tres primeras bandas a rojo, verde y azul para crear una imagen en color (este es el estilo de dibujo `MultiBandColor`). En algunos casos, es posible que desee anular esta configuración. El siguiente código intercambia la banda roja (1) y la banda verde (2):

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

En caso de que sea necesaria solo una banda para la visualización del ráster, se puede elegir el dibujo de una banda única, ya sea niveles grises o pseudocolor.

Tenemos que usar `triggerRepaint()` para actualizar el mapa y ver el resultado:

```
rlayer_multi.triggerRepaint()
```

5.3 Valores de consulta

Los valores rster ´pueden ser consultados usando el método `sample()` de la clase `QgsRasterDataProvider` . Tienes que especificar una `QgsPointXY` y el número de banda de la capa ráster que quiera consultar. El método devuelve una tupla con el valor y `True` o `False` dependiendo del resultado:

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

Otro método para consultar valores ráster es usar el método `identify()` que devuelve un objeto `QgsRasterIdentifyResult`.

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

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

```
{1: 323.0}
```

En este caso, el método `results()` devuelve un diccionario, con el índice de banda como clave, y valores de banda como valores. Por ejemplo, algo como `{1: 323.0}`

5.4 Edición de datos ráster

Puede crear una capa ráster utilizando la clase `QgsRasterBlock`. Por ejemplo, para crear un bloque ráster de 2x2 con un byte por píxel:

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

Los píxeles de trama se pueden sobrescribir gracias al método `writeBlock()`. Para sobrescribir datos ráster existentes en la posición 0,0 por el bloque 2x2:

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

CAPÍTULO 6

Usar capas vectoriales

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

from qgis.core.additions.edit import edit

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

Esta sección sumariza varias acciones que pueden ser realizadas con las capas vectoriales

La mayor parte del trabajo acá expuesto está basado en los métodos de la clase `QgsVectorLayer`.

6.1 Recuperando información sobre atributos

Puede recuperar información sobre los campos asociados a una capa vectorial llamando el método `fields()` de un objeto de la clase `QgsVectorLayer`

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

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

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

Cuando carga una capa vectorial, un campo es siempre escogido por QGIS como el `Nombre de Visualización`, mientras que el `Aviso Mapa HTML` está vacío de forma predeterminada. Con estos métodos puede fácilmente conseguir ambos:

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

```
NAME
```

Nota: Si cambia el `Nombre de Visualización` de un campo a una expresión, tiene que usar `displayExpression()` en vez de `displayField()`.

6.2 Iterando sobre la capa vectorial

a iteración de las entidades de una capa vectorial es una de las tareas más comunes. A continuación se muestra un ejemplo del código básico simple para realizar esta tarea y mostrar cierta información sobre cada característica. Se supone que la variable “”layer”” tiene un objeto `QgsVectorLayer`.

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

for feature in features:
    # retrieve every feature with its geometry and attributes
    print("Feature ID: ", feature.id())
    # fetch geometry
    # show some information about the feature geometry
    geom = feature.geometry()
    geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
    if geom.type() == QgsWkbTypes.PointGeometry:
        # the geometry type can be of single or multi type
        if geomSingleType:
            x = geom.asPoint()
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

6.3 Seleccionando objetos espaciales

En el escritorio QGIS, las entidades se pueden seleccionar de diferentes maneras: el usuario puede hacer clic en una entidad, dibujar un rectángulo en el lienzo del mapa o utilizar un filtro de expresión. Las entidades seleccionadas normalmente se resaltan en un color diferente (el valor predeterminado es el amarillo) para llamar la atención del usuario sobre la selección.

A veces puede ser útil seleccionar características mediante programación o cambiar el color predeterminado.

Para seleccionar todas las características, se puede utilizar el método `selectAll()`

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

Para seleccionar usando una expresión, utilice el método `selectByExpression()`

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

Para cambiar el color de selección puede utilizar el método `setSelectionColor()` de `QgsMapCanvas` como se muestra en el ejemplo siguiente:

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

Para agregar entidades a la lista de entidades seleccionada para una capa determinada, puede llamar a `select()` pasándole la lista de identificadores de las entidades:

```
selected_fid = []

# Get the first feature id from the layer
feature = next(layer.getFeatures())
if feature:
    selected_fid.append(feature.id())

# Add that features to the selected list
layer.select(selected_fid)
```

Para borrar la selección:

```
layer.removeSelection()
```

6.3.1 Accediendo a atributos

Los atributos pueden ser referidos por su nombre:

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

```
First feature
```

Alternativamente, se puede hacer referencia a los atributos por índice. Esto es un poco más rápido que usar el nombre. Por ejemplo, para obtener el segundo atributo:

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

```
First feature
```

6.3.2 Iterando sobre rasgos seleccionados

Si solo necesita entidades seleccionadas, puede utilizar el método `selectedFeatures()` de la capa vectorial:

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

6.3.3 Iterando sobre un subconjunto de rasgos

Si desea iterar sobre un subconjunto determinado de entidades de una capa, como las que se encuentran en un área determinada, debe agregar un objeto `QgsFeatureRequest` a la llamada de `getFeatures()`. Este es un ejemplo:

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

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

En aras de la velocidad, la intersección a menudo se realiza solo con el cuadro delimitador de la entidad. Sin embargo, hay una bandera `ExactIntersect` que se asegura de que solo se devolverán las entidades que se cruzan:

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

Con `setLimit()` puede limitar el número de entidades solicitadas. Este es un ejemplo:

```
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>
```

Si necesita un filtro basado en atributos en su lugar (o además) de uno espacial como se muestra en los ejemplos anteriores, puede crear un objeto `QgsExpression` y pasarlo al constructor `QgsFeatureRequest`. Este es un ejemplo:

```
# 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)
```

Consulte *Expresiones, Filtros y Calculando Valores* para obtener detalles sobre la sintaxis admitida por `QgsExpression`.

La solicitud se puede utilizar para definir los datos recuperados para cada entidad, por lo que el iterador devuelve todas las entidades, pero devuelve datos parciales para cada una de ellas.

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

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

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

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

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

6.4 Modificación de capas vectoriales

La mayoría de los proveedores de datos vectoriales admiten la edición de datos de capa. A veces solo admiten un subconjunto de posibles acciones de edición. Utilice la función `capabilities()` para averiguar qué conjunto de funcionalidad es compatible.

```
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 obtener una lista de todas las capacidades disponibles, consulte la documentación `API Documentation of QgsVectorDataProvider`.

Para imprimir la descripción textual de las capacidades de las capas en una lista separada por comas, puede utilizar `capabilitiesString()` como en el ejemplo siguiente:

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

Mediante el uso de cualquiera de los métodos siguientes para la edición de capas vectoriales, los cambios se confirman directamente en el almacén de datos subyacente (un archivo, una base de datos, etc.). En caso de que desee realizar solo cambios temporales, vaya a la siguiente sección que explica cómo hacer *modificaciones con la edición de cambios de búfer*.

Nota: Si está trabajando dentro de QGIS (ya sea desde la consola o desde un complemento), podría ser necesario forzar un redibujo del lienzo del mapa para ver los cambios que ha realizado en la geometría, en el estilo o en los atributos:

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

6.4.1 Añadir Entidades

Crea algunos ejemplos `QgsFeature` y pasa una lista de ellos al método del proveedor `addFeatures()`. Devolverá dos valores: resultado (`True` o `False`) y una lista de entidades agregadas (su ID lo establece el almacén de datos).

Para configurar los atributos de la entidad, puede inicializar la entidad pasando un objeto `QgsFields` (puede obtenerlo del método `fields()` de la capa vectorial) o llamar a `initAttributes()` pasando el número de campos que desea agregar.

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

6.4.2 Borrar Entidades

Para eliminar algunas entidades, solo tiene que proporcionar una lista de identificaciones de entidades.

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

6.4.3 Modificar los objetos espaciales

Es posible cambiar la geometría de la entidad o cambiar algunos atributos. En el ejemplo siguiente se cambian primero los valores de los atributos con los índices 0 y 1 y, a continuación, se cambia la geometría de la entidad.

```
fid = 100   # ID of the feature we will modify

if caps & QgsVectorDataProvider.ChangeAttributeValues:
    attrs = { 0 : "hello", 1 : 123 }
    layer.dataProvider().changeAttributeValues({ fid : attrs })

if caps & QgsVectorDataProvider.ChangeGeometries:
    geom = QgsGeometry.fromPointXY(QgsPointXY(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

Truco: Favorecer la clase QgsVectorLayerEditUtils para ediciones de solo geometría

Si solo necesita cambiar geometrías, podría considerar el uso de `QgsVectorLayerEditUtils` que proporciona algunos métodos útiles para editar geometrías (trasladar, insertar o mover vértices, etc.).

6.4.4 Modificación de capas vectoriales con un búfer de edición

Al editar vectores dentro de la aplicación QGIS, primero debe iniciar el modo de edición para una capa en particular, luego hacer algunas modificaciones y finalmente confirmar (o deshacer) los cambios. Todos los cambios que realiza no se escriben hasta que los confirma; permanecen en el búfer de edición en memoria de la capa. Es posible utilizar esta funcionalidad también de forma programática, es solo otro método para la edición de capas vectoriales que complementa el uso directo de los proveedores de datos. Use esta opción cuando proporcione algunas herramientas GUI para la edición de capas vectoriales, ya que esto permitirá al usuario decidir si confirmar / deshacer y permite el uso de deshacer/rehacer. Cuando se confirman los cambios, todos los cambios del búfer de edición se guardan en el proveedor de datos.

Los métodos son similares a los que hemos visto en el proveedor, pero se llaman en el objeto `QgsVectorLayer` en su lugar.

Para que estos métodos funcionen, la capa debe estar en modo de edición. Para iniciar el modo de edición, utilice el método `startEditing()` Para detener la edición, utilice los métodos `commitChanges()` o `rollBack()`. El primero confirmará todos los cambios en el origen de datos, mientras que el segundo los descartará y no modificará el origen de datos en absoluto.

Para averiguar si una capa está en modo de edición, utilice el método `isEditable()`.

Aquí tiene algunos ejemplos que muestran cómo utilizar estos métodos de edición.

```
from qgis.PyQt.QtCore import QVariant

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

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```
value ='My new name'
layer.changeAttributeValue(fid, fieldIndex, value)

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

Para hacer que deshacer/rehacer trabaje correctamente, las llamadas mencionadas arriba tienen que ser envueltas en los comandos undo. (Si no le importa deshacer/rehacer y desea que los cambios se almacenen inmediatamente, entonces tendrá un trabajo más fácil por *editando con proveedor de datos*.)

Así es cómo usted puede utilizar la funcionalidad de deshacer:

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

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

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

# ... more editing ...

layer.endEditCommand()
```

El método `beginEditCommand()` creará un comando interno de «activo» y registrará los cambios posteriores en la capa vectorial. Con la llamada a el comando `endEditCommand()` se inserta en la pila de deshacer y el usuario podrá deshacer/rehacerlo desde la GUI. En caso de que algo saliera mal al realizar los cambios, el método `destroyEditCommand()` quitará el comando y revertirá todos los cambios realizados mientras este comando estaba activo.

También puede utilizar la instrucción `with edit(layer)` - para encapsular la confirmación y la reversión en un bloque de código más semántico, como se muestra en el ejemplo siguiente:

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

Esto llamará automáticamente a `commitChanges()` al final. Si ocurre alguna excepción, hará `rollBack()` a todos los cambios. En caso de que se encuentre un problema dentro de `commitChanges()` (cuando el método devuelve False) se producirá una excepción `QgsEditError`.

6.4.5 Agregando y Removiendo Campos

Para agregar campos (atributos), usted necesita especificar una lista de definiciones de campo. Para la eliminación de campos sólo proporcione una lista de índices de campo.

```
from qgis.PyQt.QtCore import QVariant

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

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

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

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

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

Después de agregar o quitar campos en el proveedor de datos, los campos de la capa deben actualizarse porque los cambios no se propagan automáticamente.

```
layer.updateFields()
```

Truco: Guarde directamente los cambios usando el comando basado en `with`

Usando `with edit(layer):` los cambios serán automàticamente enviados a `commitChanges()` al final. si ocurre alguna excepción, serán `rollBack()` todos los cambios. Ver *Modificación de capas vectoriales con un búfer de edición*.

6.5 Usar índice espacial

Los índices espaciales pueden mejorar drásticamente el rendimiento del código si necesita realizar consultas frecuentes en una capa vectorial. Imagine, por ejemplo, que está escribiendo un algoritmo de interpolación, y que para una ubicación determinada necesita conocer los 10 puntos más cercanos de una capa de puntos, con el fin de utilizar esos puntos para calcular el valor interpolado. Sin un índice espacial, la única manera de que QGIS encuentre esos 10 puntos es calcular la distancia desde todos y cada uno de los puntos hasta la ubicación especificada y luego comparar esas distancias. Esto puede ser una tarea que consume mucho tiempo, especialmente si necesita repetirse para varias ubicaciones. Si existe un índice espacial para la capa, la operación es mucho más efectiva.

Piense en una capa sin un índice espacial como una guía telefónica en la que los números de teléfono no se ordenan ni indexan. La única manera de encontrar el número de teléfono de una persona determinada es leer desde el principio hasta que lo encuentres.

Los índices espaciales no se crean de forma predeterminada para una capa vectorial QGIS, pero puede crearlos fácilmente. Esto es lo que tienes que hacer:

- crear índice espacial usando la clase `QgsSpatialIndex`:

  ```
  index = QgsSpatialIndex()
  ```

- agregar entidades al índice — el índice toma el objeto `QgsFeature` y lo agrega a la estructura de datos interna. Puede crear el objeto manualmente o usar uno de una llamada anterior al método `getFeatures()` del proveedor.

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

- alternativamente, puede cargar todas las entidades de una capa a la vez utilizando la carga masiva

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

- Una vez que el índice espacial se llena con algunos valores, puede realizar algunas consultas

```
# returns array of feature IDs of five nearest features
nearest = index.nearestNeighbor(QgsPointXY(25.4, 12.7), 5)

# returns array of IDs of features which intersect the rectangle
intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
```

También puede usar el índice espacial `QgsSpatialIndexKDBush`. Este índice es similar al *estándar* `QgsSpatialIndex` pero:

- soporta **solo** entidades de único punto
- es **estática** (no se pueden agregar entidades adicionales al índice después de la construcción)
- es **¡mas rápida!**
- permite la recuperación directa de los puntos de la entidad original, sin requerir solicitudes de entidades adicionales
- admite búsquedas verdaderas *basadas en la distancia*, es decir, devuelve todos los puntos dentro de un radio desde un punto de búsqueda

6.6 La clase QgsVectorLayerUtils

La clase `QgsVectorLayerUtils` contiene algunos métodos muy útiles que puede utilizar con capas vectoriales.

Por ejemplo el método `createFeature()` prepara una `QgsFeature` para ser agregada a una capa vectorial manteniendo todas las eventuales restricciones y valores predeterminados de cada campo:

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

El método `getValues()` le permite obtener rápidamente los valores de un campo o expresión:

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

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

6.7 Creación de capas vectoriales

Hay varias maneras de generar un dataset de capa vectorial:

- la clase `QgsVectorFileWriter` que guarda la capa vectorial entera o creando una instancia de la clase y enviando llamadas a meth:*addFeature() <qgis.core.QgsVectorFileWriter.addFeature*>. Esta clase soporta todos los formatos vectoriales que soporta GDAL s(GeoPackage, Shapefile, GeoJSON, KML y otros).
- la clase `QgsVectorLayer`: crea una instancia de un proveedor de datos que interpreta la ruta seleccionada (url) de la fuente de datos para conectar y acceder a los datos. Se puede usar para crear capas temporales, basadas en la memoria (`memoria`) y conectar a conjunto de datos vectoriales GDAL (`ogr`), bases de datos (`postgres`, `spatialite`, `mysql`, `mssql`) y más (`wfs`, `gpx`, `delimitedtext`...).

6.7.1 Desde una instancia de `QgsVectorFileWriter`

```
# SaveVectorOptions contains many settings for the writer process
save_options = QgsVectorFileWriter.SaveVectorOptions()
transform_context = QgsProject.instance().transformContext()
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
                                                  "testdata/my_new_file.gpkg",
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
else:
  print(error)
```

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "ESRI Shapefile"
save_options.fileEncoding = "UTF-8"
transform_context = QgsProject.instance().transformContext()
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
                                                  "testdata/my_new_shapefile",
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
else:
  print(error)
```

```
# Write to an ESRI GDB file
save_options = QgsVectorFileWriter.SaveVectorOptions()
save_options.driverName = "FileGDB"
# if no geometry
save_options.overrideGeometryType = QgsWkbTypes.Unknown
save_options.actionOnExistingFile = QgsVectorFileWriter.CreateOrOverwriteLayer
save_options.layerName = 'my_new_layer_name'
transform_context = QgsProject.instance().transformContext()
gdb_path = "testdata/my_example.gdb"
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
                                                  gdb_path,
                                                  transform_context,
                                                  save_options)
if error[0] == QgsVectorFileWriter.NoError:
  print("success!")
else:
  print(error)
```

También puede convertir campos para que sean compatibles con formatos diferentes usando `FieldValueConverter`. Por ejemplo, para convertir tipos de variable matriz (por ejemplo, en Postgres) en un tipo texto, puede hacer lo siguiente:

```
LIST_FIELD_NAME = 'xxxx'

class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

  def __init__(self, layer, list_field):
    QgsVectorFileWriter.FieldValueConverter.__init__(self)
    self.layer = layer
    self.list_field_idx = self.layer.fields().indexFromName(list_field)

  def convert(self, fieldIdxInLayer, value):
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

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

También se puede especificar un CRS de destino — si se pasa una instancia válida de `QgsCoordinateReferenceSystem` como cuarto parámetro, la capa se transforma a ese CRS.

Para obtener nombres de controladores válidos, llame al método `supportedFiltersAndFormats()` o consulte los formatos admitidos por OGR — debe pasar el valor en la columna «Código» como el nombre del controlador.

Opcionalmente, puede establecer si desea exportar solo las entidades seleccionadas, pasar más opciones específicas del controlador para la creación o indicar al escritor que no cree atributos… Hay una serie de otros parámetros (opcionales); consulte la documentación de `QgsVectorFileWriter` para más detalles.

6.7.2 Directamente desde las funciones

```
from qgis.PyQt.QtCore import QVariant

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

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

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

writer = QgsVectorFileWriter.create(
  "testdata/my_new_shapefile.shp",
  fields,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  QgsWkbTypes.Point,
  crs,
  transform_context,
  save_options
)

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

# add a feature
fet = QgsFeature()

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

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

6.7.3 Desde una instancia de `QgsVectorLayer`

Entre todos los proveedores de datos admitidos por la clase `QgsVectorLayer`, vamos a centrarnos en las capas basadas en memoria. Proveedor de memoria está destinado a ser utilizado principalmente por plugins o desarrolladores de aplicaciones de 3as partes. No almacena datos en el disco, lo que permite a los desarrolladores utilizarlos como un backend rápido para algunas capas temporales.

El proveedor admite los campos string, int y double.

El proveedor de memoria también admite la indexación espacial, que se habilita llamando a la función `createSpatialIndex()` del proveedor. Una vez creado el índice espacial, podrá recorrer iterando sobre las entidades dentro de regiones más pequeñas más rápido (ya que no es necesario atravesar todas las entidades, solo las del rectángulo especificado)..

Un proveedor de memoria se crea pasando `"memory"` como la cadena del proveedor al constructor `QgsVectorLayer`.

El constructor también toma un URI que define el tipo de geometría de la capa, uno de: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, `"MultiPolygon"` o `"None"`.

El URI también puede especificar el sistema de referencia de coordenadas, los campos y la indexación del proveedor de memoria en el URI. La sintaxis es:

crs=definición
: Especifica el sistema de referencia de coordenadas, donde la definición puede ser cualquiera de las formas aceptadas por `QgsCoordinateReferenceSystem.createFromString()`

index=yes
: Especifica que el proveedor utilizará un índice espacial

campo
: Especifica un atributo de la capa. El atributo tiene un nombre y, opcionalmente, un tipo (entero, doble o cadena), longitud y precisión. Puede haber múltiples definiciones de campo

El siguiente ejemplo de una URI incorpora todas estas opciones

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

El siguiente código de ejemplo ilustra como crear y rellenar un proveedor de memoria

```
from qgis.PyQt.QtCore import QVariant

```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])

# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()
```

Finalmente, vamos a comprobar si todo salió bien

```
# show some stats
print("fields:", len(pr.fields()))
print("features:", pr.featureCount())
e = vl.extent()
print("extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum())

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

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

6.8 Apariencia (Simbología) de capas vectoriales

Cuando una capa vectorial se representa, la apariencia de los datos se indica por **renderer** y **símbolos** asociados a la capa. Los símbolos son clases que se encargan del dibujo de la representación visual de las entidades, mientras que los renderizadores determinan qué símbolo se utilizará para una entidad determinada.

El renderizador para una capa determinada se puede obtener como se muestra a continuación:

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

Y con esa referencia, vamos a explorar un poco

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

```
Type: singleSymbol
```

Hay varios tipos de renderizadores conocidos disponibles en la biblioteca principal de QGIS:

Tipo	Clase	Descripción
singleSymbol	QgsSingleSymbolRendere	Representa todas las entidades con el mismo símbolo
categorizedSym	QgsCategorizedSymbolRe	Representa entidades utilizando un símbolo diferente para cada categoría
graduatedSymb	QgsGraduatedSymbolRend	Representa entidades utilizando un símbolo diferente para cada rango de valores

También puede haber algunos tipos de presentadores personalizados, por lo que nunca haga una suposición de que solo hay estos tipos. Puede consultar el `QgsRendererRegistry` de la aplicación para encontrar los presentadores disponibles actualmente:

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

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

Es posible obtener un volcado del contenido de presentador en forma de texto — puede ser útil para la depuración

```
renderer.dump()
```

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

6.8.1 Representador de Símbolo Único

Puede obtener el símbolo utilizado para la representación llamando al método `symbol()` y cambiarlo con el método :meth:`setSymbol() <qgis.core.QgsSingleSymbolRenderer.setSymbol>`(nota para desarrolladores de C++: el renderizado toma posesión del símbolo.)

Puede cambiar el símbolo utilizado por una capa vectorial determinada llamando a `setSymbol()` pasando una instancia de la instancia de símbolo apropiada. Los símbolos para las capas de *point*, *line* y *polygon* pueden ser creadas llamando a la función `createSimple()` de las clases correspondientes `QgsMarkerSymbol`, `QgsLineSymbol` y `QgsFillSymbol`.

El diccionario pasado a `createSimple()` establece las propiedades de estilo del símbolo.

Por ejemplo, puede reemplazar el símbolo utilizado por una capa **point** particular llamando a `setSymbol()` pasando una instancia de `QgsMarkerSymbol`, como en el siguiente ejemplo de código:

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

`name` indica la forma del marcador, y puede ser cualquiera de los siguientes:

- `circle`
- `square`
- `cross`
- `rectangle`
- `diamond`
- `pentagon`
- `triangle`
- `equilateral_triangle`

- star
- regular_star
- arrow
- filled_arrowhead
- x

Para obtener la lista completa de propiedades de la primera capa de símbolos de una instancia de símbolo, puede seguir el código de ejemplo:

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

```
{'angle': '0', 'cap_style': 'square', 'color': '255,0,0,255', 'horizontal_anchor_
↪point': '1', 'joinstyle': 'bevel', 'name': 'square', 'offset': '0,0', 'offset_
↪map_unit_scale': '3x:0,0,0,0,0,0', 'offset_unit': 'MM', 'outline_color': '35,35,
↪35,255', 'outline_style': 'solid', 'outline_width': '0', 'outline_width_map_unit_
↪scale': '3x:0,0,0,0,0,0', 'outline_width_unit': 'MM', 'scale_method': 'diameter',
↪ 'size': '2', 'size_map_unit_scale': '3x:0,0,0,0,0,0', 'size_unit': 'MM',
↪'vertical_anchor_point': '1'}
```

Esto puede ser útil si desea modificar algunas propiedades:

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

6.8.2 Representador de símbolo categorizado

Al utilizar un representador categorizado, puede consultar y establecer el atributo que se utiliza para la clasificación: utilice los métodos `classAttribute()` y `setClassAttribute()`.

Para obtener una lista de categorías

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

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

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

Donde `value()` es el valor utilizado para la discriminación entre categorías, `label()` es el texto utilizado para la descripción de la categoría y el método `symbol()` devuelve el símbolo asignado.

El renderizador suele almacenar también el símbolo original y la rampa de color que se utilizaron para la clasificación: métodos `sourceColorRamp()` y `sourceSymbol()`.

6.8.3 Renderizador de símbolo graduado

Este representador es muy similar al representador de símbolos categorizados descrito anteriormente, pero en lugar de un valor de atributo por clase trabaja con rangos de valores y, por lo tanto, solo se puede utilizar con atributos numéricos.

Para obtener más información sobre los rangos utilizados en el renderizador

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

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

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

puedes volver a usar los métodos `classAttribute()` (para encontrar el nombre del atributo de clasificación), `sourceSymbol()` y `sourceColorRamp()` . Adicionalmente está el método `mode()` que determina como son creados los rangos: usando intervalos iguales, cuanties o algún otro método.

Si desea crear su propio renderizador de símbolos graduados, puede hacerlo como se ilustra en el siguiente fragmento de ejemplo (que crea una sencilla disposición de dos clases)

```
from qgis.PyQt import QtGui

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

(continúe en la próxima página)

(proviene de la página anterior)

```
    ↪"EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
```

6.8.4 Trabajo con Símbolos

Para la representación de símbolos, hay esta clase base `QgsSymbol` con tres clases derivadas:

- `QgsMarkerSymbol` — para entidades de punto
- `QgsLineSymbol` — para entidades de línea
- `QgsFillSymbol` — para entidades de polígono

Cada símbolo consta de una o más capas de símbolos (clases derivadas de `QgsSymbolLayer`). Las capas de símbolos hacen la representación real, la propia clase de símbolo sirve sólo como un contenedor para las capas de símbolo.

Teniendo una instancia de un símbolo (por ejemplo, de un renderizador), es posible explorarlo: el método `type()` dice si es un marcador, línea o símbolo de relleno . Hay un método `dump()` que devuelve una breve descripción del símbolo. Para obtener una lista de capas de símbolos:

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

```
0: SimpleMarker
```

Para averiguar el color del símbolo use el método `color()` y `setColor()` para cmabiar su color. Con los símbolos de marcador, además, puede consultar el tamaño y la rotación del símbolo con los métodos `size()` y `angle()` . Para símbolos lineales el método `width()` devuelve la anchura de la línea.

De forma predeterminada el tamaño y ancho están en milímetros, los ángulos en grados.

Trabajando con capas de símbolos

Como se dijo antes, las capas de símbolo (subclases de `QgsSymbolLayer`) determinar la apariencia de las entidades. Hay varias clases de capas de símbolos básicas para uso general. Es posible implementar nuevos tipos de capas de símbolos y así personalizar arbitrariamente cómo se renderizarán las entidades. El método `layerType()` identifica de forma única la clase de capa de símbolo — las básicas y predeterminadas son los tipos de capas de símbolo `` SimpleMarker “”, `` SimpleLine “” y `` SimpleFill “”.

identifica de forma única la clase de capa de símbolo — las básicas y predeterminadas son los tipos de capas de símbolo `` SimpleMarker “”, `` SimpleLine “” y `` SimpleFill “”.

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

```
AnimatedMarker
EllipseMarker
FilledMarker
FontMarker
GeometryGenerator
```

(continúe en la próxima página)

(proviene de la página anterior)

```
MaskMarker
RasterMarker
SimpleMarker
SvgMarker
VectorField
```

La clase `QgsSymbolLayerRegistry` gestiona una base de datos de todos los tipos de capas de símbolos disponibles.

Para acceder a los datos de la capa de símbolos, use su método `properties()` que devuelve un diccionario de propiedades clave-valor que determina la apariencia. Cada tipo de capa de símbolo tiene un conjunto específico de propiedades que utiliza. Además, existen los métodos genéricos `color()`, `size()`, `angle()` y `width()`, con sus contrapartes setter. Por supuesto, el tamaño y el ángulo solo están disponibles para las capas de símbolo de marcador y el ancho para las capas de símbolo de línea.

Crear tipos de capas de símbolos personalizados

Imagine que le gustaría personalizar la forma en que se procesan los datos. Puede crear su propia clase de capa de símbolo que dibujará las entidades exactamente como lo desee. Aquí hay un ejemplo de un marcador que dibuja círculos rojos con un radio especificado

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

class FooSymbolLayer(QgsMarkerSymbolLayer):

  def __init__(self, radius=4.0):
      QgsMarkerSymbolLayer.__init__(self)
      self.radius = radius
      self.color = QColor(255,0,0)

  def layerType(self):
     return "FooMarker"

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

  def startRender(self, context):
    pass

  def stopRender(self, context):
      pass

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

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

El `layerType()` el método determina el nombre de la capa de símbolo; tiene que ser único entre todas las capas de símbolos. El método `properties()` es usada para la persistencia de sus atributos. El método `clone()` debe devolver una copia de la capa de símbolo con todos los atributos exactamente iguales. Finalmente, existen métodos de renderizado: `startRender()` se llama antes de renderizar la primera entidad, `stopRender()` cuando el renderizado está terminado, y `renderPoint()` es llamado para hacer el renderizado. Las coordenadas de los puntos ya están transformadas a las coordenadas de salida.

Para polilíneas y polígonos, la única diferencia estaría en el método de representación: que usaría `renderPolyline()` que recibe una lista de líneas, mientras que :meth:`renderPolygon() <qgis.core.QgsFillSymbolLayer.renderPolygon> `recibe una lista de puntos en el anillo exterior como primer parámetro y una lista de anillos interiores (o Ninguno) como segundo parámetro.

Por lo general, es conveniente agregar una GUI para configurar los atributos del tipo de capa de símbolo para permitir a los usuarios personalizar la apariencia: en el caso de nuestro ejemplo anterior, podemos permitir que el usuario establezca el radio del círculo. El siguiente código implementa dicho widget

```
from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

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

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

    def symbolLayer(self):
        return self.layer

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

Este widget se puede incrustar en el cuadro de diálogo de propiedades del símbolo. Cuando se selecciona el tipo de capa de símbolo en el cuadro de diálogo de propiedades de símbolo, crea una instancia de la capa de símbolo y una instancia del widget de capa de símbolo. Luego llama al método `setSymbolLayer()` para asignar la capa de símbolo al widget. En ese método, el widget debería actualizar la interfaz de usuario para reflejar los atributos de la capa de símbolo. El método `symbolLayer()` se usa para recuperar la capa de símbolo nuevamente mediante el diálogo de propiedades para usarla para el símbolo.

En cada cambio de atributos, el widget debe emitir la señal `changed()` para permitir que el diálogo de propiedades actualice la vista previa del símbolo.

Ahora solo nos falta el pegamento final: hacer que QGIS esté al tanto de estas nuevas clases. Esto se hace agregando la capa de símbolo al registro. También es posible utilizar la capa de símbolo sin agregarla al registro, pero algunas funciones no funcionarán: p. Ej. carga de archivos de proyecto con capas de símbolo personalizadas o incapacidad para editar los atributos de la capa en la GUI.

Tendremos que crear metadatos para la capa de símbolo.

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

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

  def __init__(self):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    super().__init__("FooMarker", "My new Foo marker", QgsSymbol.Marker)

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

fslmetadata = FooSymbolLayerMetadata()
```

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

Debe pasar el tipo de capa (el mismo que devuelve la capa) y el tipo de símbolo (marcador / línea / relleno) al constructor de la clase principal. El método `createSymbolLayer()` se encarga de crear una instancia de la capa de símbolo con los atributos especificados en el diccionario *props*. Y esta el método `createSymbolLayerWidget()` que devuelve el widget de configuración para este tipo de capa de símbolo.

El último pase es adicionar esta capa símbolo al registro — y estamos listos.

6.8.5 Crear Renderizados personalizados

Puede ser útil crear una nueva implementación de renderizador si desea personalizar las reglas sobre cómo seleccionar símbolos para renderizar características. Algunos casos de uso en los que desearía hacerlo: el símbolo se determina a partir de una combinación de campos, el tamaño de los símbolos cambia según la escala actual, etc.

El siguiente código muestra un renderizador personalizado simple que crea dos símbolos de marcador y elige aleatoriamente uno de ellos para cada objeto.

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


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

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

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

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

  def usedAttributes(self, context):
    return []

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

El constructor de la clase padre `QgsFeatureRenderer` necesita un nombre de renderizador (que tiene que ser único entre los renderizadores). El método `symbolForFeature()` es el que decide qué símbolo se utilizará para una entidad en particular. `startRender()` y `stopRender()` se encargan de la inicialización/finalización de la

representación de símbolos. El método `usedAttributes()` puede devolver una lista de nombres de campo que el renderizador espera que esté presente. Finalmente, el método `clone()` debería devolver una copia del renderizador.

Al igual que con las capas de símbolos, es posible adjuntar una GUI para la configuración del renderizador. Tiene que derivarse de `QgsRendererWidget`. El siguiente código de muestra crea un botón que permite al usuario establecer el primer símbolo

```
from qgis.gui import QgsRendererWidget, QgsColorButton


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

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

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

El constructor recibe instancias de la capa activa (`QgsVectorLayer`), el estilo global (`QgsStyle`) y el renderizador actual. Si no hay renderizador o el renderizador tiene un tipo diferente, será reemplazado por nuestro nuevo renderizador, de lo contrario usaremos el renderizador actual (que ya tiene el tipo que necesitamos). El contenido del widget debe actualizarse para mostrar el estado actual del renderizador. Cuando se acepta el cuadro de diálogo del renderizador, se llama al método del widget : meth:*renderer() <qgis.gui.QgsRendererWidget.renderer>* para obtener el renderizador actual — se asignará a la capa.

El último bit que falta son los metadatos del renderizador y el registro en el registro; de lo contrario, la carga de capas con el renderizador no funcionará y el usuario no podrá seleccionarlo de la lista de renderizadores. Terminemos nuestro ejemplo de RandomRenderer

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

class RandomRendererMetadata(QgsRendererAbstractMetadata):

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

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

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

rrmetadata = RandomRendererMetadata()
```

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

Del mismo modo que con las capas de símbolo, el constructor de metadatos abstracto espera el nombre del renderizador, el nombre visible para los usuarios y, opcionalmente, el nombre del icono del renderizador. El método `createRenderer()` pasa una instancia `QDomElement` que puede usarse para restaurar el estado del renderizador desde el árbol DOM. El método `createRendererWidget()` crea el widget de configuración. No tiene que estar presente o puede devolver `Ninguno` si el renderizador no viene con GUI.

Para asociar un icono con el renderizador, puede asignarlo en el constructor `QgsRendererAbstractMetadata` como tercer argumento (opcional) — el constructor de la clase base en la función RandomRendererMetadata `__init__()` se convierte en

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

El icono también se puede asociar en cualquier momento posterior mediante el método `setIcon()` de la clase de metadatos. El icono se puede cargar desde un archivo (como se muestra arriba) o se puede cargar desde un recurso Qt (PyQt5 incluye el compilador .qrc para Pitón).

6.9 Más Temas

PENDIENTE:

- crear/modificar símbolos
- trabajandocon estilo (`QgsStyle`)
- trabajando con rampas de color (`QgsColorRamp`)
- explorar la capa de símbolo y los registros de renderizado

CAPÍTULO 7

Manejo de Geometría

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

Los puntos, las cadenas de líneas y los polígonos que representan una característica espacial se conocen comúnmente como geometrías. En QGIS están representados con la clase `QgsGeometry`.

A veces una geometría es realmente una colección simple (partes simples) geométricas. Tal geometría se llama geometría de múltiples partes. Si contiene un tipo de geometría simple, lo llamamos un punto múltiple, lineas múltiples o polígonos múltiples. Por ejemplo, un país consiste en múltiples islas que se pueden representar como un polígono múltiple.

Las coordenadas de las geometrías pueden estar en cualquier sistema de referencia de coordenadas (SRC). Cuando extrae características de una capa, las geometrías asociadas tendrán sus coordenadas en el SRC de la capa.

La descripción y las especificaciones de todas las geometrías posibles, la construcción y las especificaciones están disponibles en los Estándares de Acceso a Funciones Simples de OGC para detalles avanzados.

7.1 Construcción de Geometría

PyQGIS proporciona varias opciones para crear una geometría:

- desde coordenadas

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

Las coordenadas son dadas usando la clase `QgsPoint` o la clase `QgsPointXY`. La diferencia entre estas clases es que soporten `QgsPoint` dimensione M y Z.

Una polilínea (cadena lineal) es representada por una lista de puntos.

Un polígono está representado por una lista de anillos lineales (es decir, cadenas de líneas cerradas). El primer anillo es el anillo exterior (límite), los anillos subsiguientes opcionales son agujeros en el polígono. Tenga en cuenta que, a diferencia de algunos programas, QGIS cerrará el anillo por usted, por lo que no es necesario duplicar el primer punto como el último.

Las geometrías multi-parte van un nivel más allá: multi-punto es una lista de puntos, multi-linea es una lista de polilíneas y multi-polígono es una lista de polígonos.

- desde well-known text (WKT)

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

- desde well-known binary (WKB)

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

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

7.2 Acceso a Geometría

Primero, debe averiguar el tipo de geometría. El método `wkbType()` es el que se va a utilizar. Devuelve un valor de la enumeración `QgsWkbTypes.Type`.

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

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

```
print(gLine.type())
# output: 'GeometryType.Line'
```

Puede usar la función `displayString()` para obtener un tipo de geometría legible por humanos.

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

También hay una función de ayuda `isMultipart()` para saber si una geometría es multiparte o no.

Para extraer información de una geometría, existen funciones de acceso para cada tipo de vector. A continuación, se muestra un ejemplo sobre cómo utilizar estos accesos:

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

Nota: Las tuplas (x, y) no son tuplas reales, son objetos `QgsPoint`, los valores son accesibles con los métodos `x()` y `y()`.

Para geometrías multiparte, existen funciones de acceso similares: `asMultiPoint()`, `asMultiPolyline()` y `asMultiPolygon()`.

Es posible iterar sobre todas las partes de una geometría, independientemente del tipo de geometría. P.ej.

```
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)
```

También es posible modificar cada parte de la geometría usando el método `QgsGeometry.parts()`.

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

(continúe en la próxima página)

(proviene de la página anterior)

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

```
MultiPoint ((-10334728.12541878595948219 -5360106.25905461423099041),(-10462135.
↪16126426123082638 -5217485.4735023295506835),(-10589399.84444035589694977 -
↪5072021.45942386891692877))
```

7.3 Geometría predicados y Operaciones

QGIS utiliza la biblioteca GEOS para operaciones de geometría avanzadas como predicados de geometría (`contains()`, `intersects()`, ...) y hacer operaciones (`combine()`, `difference()`, ...). También puede calcular propiedades geométricas de geometrías, como área (en el caso de polígonos) o longitudes (para polígonos y líneas).

Veamos un ejemplo que combina iterar sobre las entidades en una capa determinada y realizar algunos cálculos geométricos basados en sus geometrías. El siguiente código calculará e imprimirá el área y el perímetro de cada país en la capa de `países` dentro de nuestro proyecto tutorial QGIS.

El siguiente código asume que `layer` es un obleto `QgsVectorLayer` que tiene el tipo de entidad Polígono.

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

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

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

```
Zambia
Area:  62.82279065343119
Perimeter:  50.65232014052552
Zimbabwe
Area:  33.41113559136517
Perimeter:  26.608288555013935
```

Ahora ha calculado e impreso las áreas y perímetros de las geometrías. Sin embargo, puede notar rápidamente que los valores son extraños. Esto se debe a que las áreas y los perímetros no tienen en cuenta el CRS cuando se calculan con los métodos `area()` y `length()` desde la clase `QgsGeometry`. Para un cálculo de área y distancia más potente, la clase `QgsDistanceArea` se puede utilizar, que puede realizar cálculos basados en elipsoides:

El siguiente código asume que `layer` es un obleto `QgsVectorLayer` que tiene el tipo de entidad Polígono.

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

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

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

```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

```
Zambia
Perimeter (m): 5539361.250294601
Area (m2): 751989035032.9031
Area (km2): 751989.0350329031
Zimbabwe
Perimeter (m): 2865021.3325076113
Area (m2): 389267821381.6008
Area (km2): 389267.8213816008
```

Alternativamente, puede querer saber la distancia entre dos puntos.

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

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

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

Puede encontrar muchos ejemplos de algoritmos que se incluyen en QGIS y utilizan estos métodos para analizar y transformar los datos vectoriales. Aquí hay algunos enlaces al código de algunos de ellos.

- Distancia y área usando `QgsDistanceArea` class: Distance matrix algorithm
- Lines to polygons algorithm

CAPÍTULO 8

Soporte de Proyecciones

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

8.1 Sistemas de coordenadas de referencia

Sistemas de Coordenadas de Referencia (SRC) son encapsulados por la clase `QgsCoordinateReferenceSystem`. Las instancias de esta clase se pueden crear de varias formas diferentes:

- especificar SRC por su ID

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

```
True
```

  QGIS admite diferentes identificadores de SRC con los siguientes formatos:

  - `EPSG:<code>` — ID asignada por la organización EPSG - manejada con `createFromOgcWms()`
  - `POSTGIS:<srid>`— ID usada en bases de datos PostGIS - manejada con `createFromSrid()`
  - `INTERNAL:<srsid>` — ID usada en la base de datos QGIS - manejada con `createFromSrsId()`
  - `PROJ:<proj>` - manejada con `createFromProj()`
  - `WKT:<wkt>` - manejada con `createFromWkt()`

Si no se especifica ningún prefijo, se asume la definición de WKT.

- especificar SRC por su well-known text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.
↪257223563]],' \
      'PRIMEM["Greenwich", 0.0], UNIT["degree",0.0174532925199433295],' \
      'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
print(crs.isValid())
```

```
True
```

- cree un SRC no válido y luego use una de las funciones `crear*` para inicializarlo. En el siguiente ejemplo usamos una cadena Proj para inicializar la proyección.

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

```
True
```

Es aconsejable comprobar si la creación (es decir, la búsqueda en la base de datos) del SRC se ha realizado correctamente: `isValid()` debe devolver `True`.

Tenga en cuenta que para la inicialización de sistemas de referencia espacial, QGIS necesita buscar los valores apropiados en su base de datos interna `srs.db`. Por lo tanto, en caso de que cree una aplicación independiente, debe configurar las rutas correctamente con `QgsApplication.setPrefixPath()`, de lo contrario, no podrá encontrar la base de datos. Si está ejecutando los comandos desde la consola de QGIS Python o desarrollando un complemento, no le importa: todo ya está configurado para usted.

Acceso a la información del sistema de referencia espacial:

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

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

Salida:

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

8.2 Transformación SRC

Puede realizar la transformación entre diferentes sistemas de referencia espacial utilizando la clase `QgsCoordinateTransform`. La forma más fácil de usarlo es crear un SRC de origen y destino y construir una instancia `QgsCoordinateTransform` con ellos y el proyecto actual. Luego simplemente llame repetidamente `transform()` function para hacer la transformación. De forma predeterminada, realiza una transformación hacia adelante, pero también es capaz de realizar una transformación inversa.

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

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

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

Salida:

```
Transformed point: <QgsPointXY: POINT(832713.79873844375833869 553423.
↪98688333143945783)>
Transformed back: <QgsPointXY: POINT(18 4.9999999999999911)>
```

CAPÍTULO 9

Usando el Lienzo de Mapa

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

from qgis.PyQt.QtCore import Qt, QRectF

from qgis.PyQt.QtWidgets import QMenu

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

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

El widget del lienzo del mapa es probablemente el widget más importante dentro de QGIS porque muestra el mapa integrado de capas de mapas superpuestos y permite la interacción con el mapa y las capas. El lienzo muestra siempre una parte del mapa definido por el alcance del lienzo actual. La interacción se realiza mediante el uso de **herramientas de mapa**: hay herramientas para desplazamiento, zum, la identificación de las capas, de medida, para editar vectores y otros. Al igual que en otros programas de gráficos, siempre hay una herramienta activa y el usuario puede cambiar entre las herramientas disponibles.

El lienzo del mapa está implementado con la clase `QgsMapCanvas` en el módulo qgis.gui . La implementación

se basa en el marco Qt Graphics View. Este marco generalmente proporciona una superficie y una vista donde se colocan elementos gráficos personalizados y el usuario puede interactuar con ellos. Asumiremos que está lo suficientemente familiarizado con Qt para comprender los conceptos de escena, vista y elementos gráficos. Si no es así, lea la descripción general del marco. <https://doc.qt.io/qt-5/graphicsview.html>`_.

Cada vez que el mapa se ha desplazado, ampliado / reducido (o alguna otra acción que desencadena una actualización), el mapa se representa nuevamente dentro de la extensión actual. Las capas se representan en una imagen (utilizando la clase `QgsMapRendererJob`) y esa imagen se muestra en el lienzo. La clase `QgsMapCanvas` También controla la actualización del mapa representado. Además de este elemento que actúa como fondo, puede haber más **elementos de lienzo de mapas**.

Los elementos típicos del lienzo del mapa son bandas elásticas (utilizadas para medir, editar vectores, etc.) o marcadores de vértice. Los elementos del lienzo generalmente se usan para dar retroalimentación visual a las herramientas de mapa, por ejemplo, al crear un nuevo polígono, la herramienta de mapa crea un elemento de lienzo con banda elástica que muestra la forma actual del polígono. Todos los elementos del lienzo del mapa son subclases de `QgsMapCanvasItem` que agrega más funcionalidad a los objetos básicos `QGraphicsItem`.

Para resumir, la arquitectura del lienzo de mapa consiste en tres conceptos:

- lienzo de mapa — para la visualización del mapa
- Los elementos de lienzo de mapa — los elementos adicionales que se pueden desplegar en un lienzo de mapa
- herramientas de mapa — para interactuar con el lienzo del mapa

9.1 Lienzo de mapa insertado

Map Canvas es un widget como cualquier otro widget de Qt, por lo que usarlo es tan simple como crearlo y mostrarlo.

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

Esto produce una ventana independiente con el lienzo de mapa. Puede también ser incrustado en un widget existente o ventana. Al utilizar archivo ui y Qt Designer, coloque un `QWidget` sobre el formulario y promuévalo a una nueva clase: establezca `QgsMapCanvas` como nombre de clase y `qgis.gui` como archivo de encabezado. La utilidad `pyuic5` se hará cargo de ella. Esta es una forma conveniente de incrustar el lienzo. La otra posibilidad es escribir manualmente el código para construir el lienzo del mapa y otros widgets (como hijos de una ventana principal o diálogo) y crea un diseño.

Por defecto, el lienzo de mapa tiene un fondo negro y no utiliza anti-aliasing. Para establecer el fondo blanco y habilitar el anti-aliasing para suavisar la presentación

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

(En caso de que se esté preguntando, `Qt` viene del modulo `PyQt.QtCore` y `Qt.white` es uno de lo que predefine las instancias `QColor`.)

Ahora es el momento de agregar algunas capas del mapa. Primero abriremos una capa y la agregaremos al proyecto actual. Luego estableceremos la extensión del lienzo y la lista de capas para el lienzo.

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

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

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

```

(continúe en la próxima página)

(proviene de la página anterior)

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

Después de ejecutar estos comandos, el lienzo debe mostrar la capa que se ha cargado.

9.2 Bandas elásticas y marcadores de vértices

Para mostrar algunos datos adicionales en la parte superior del mapa en el lienzo, utilice los elementos del lienzo de mapa. Es posible crear clases de elementos del lienzo personalizada (cubiertas más abajo), sin embargo, hay dos clases de elementos de lienzo útiles para mayor comodidad `QgsRubberBand` para dibujar polilíneas o polígonos, y `QgsVertexMarker` para dibujar puntos. Ambos trabajan con coordenadas de mapa, por lo que la figura se mueve/se escala de forma automática cuando el lienzo está siendo desplazado o haciendo zum.

Para mostrar una polilínea:

```
r = QgsRubberBand(canvas, QgsWkbTypes.LineGeometry)  # line
points = [QgsPoint(-100, 45), QgsPoint(10, 60), QgsPoint(120, 45)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Para mostrar un polígono

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

Tenga en cuenta que los puntos de polígonos no es una lista simple: de hecho, es una lista de anillos que contienen lista de anillos del polígono: el primer anillo es el borde exterior, anillos adicionales (opcional) corresponden a los agujeros en el polígono.

Las bandas elásticas permiten algún tipo de personalizacion, es decir, para cambiar su color o ancho de línea

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

Los elementos del lienzo están vinculados a la escena del lienzo. Para ocultarlos temporalmente (y mostrarlos nuevamente), use el combo `hide()` and `show()`. Para eliminar completamente el elemento, debe eliminarlo de la escena del lienzo.

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

(en C++ es posible simplemente eliminar el elemento, sin embargo en Python `del r` sería simplemente suprimir la referencia y el objeto aún existirá ya que es propiedad del lienzo)

La banda de goma también se puede usar para dibujar puntos, pero la clase `QgsVertexMarker` Es más adecuada para esto (`QgsRubberBand` solo dibujaría un rectángulo alrededor del punto deseado).

Puede usar el marcador de vértices así:

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

Esto dibujará una cruz roja en la posición **[10,45]**. Es posible personalizar el tipo de icono, tamaño, color y ancho del lápiz.

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

Para ocultar temporalmente los marcadores de vértice y eliminarlos del lienzo, utilice los mismos métodos que para las gomas elásticas.

9.3 Utilizar las herramientas del mapa con el lienzo

El siguiente ejemplo construye una ventana que contiene un lienzo de mapa y herramientas de mapa básicas para la panorámica y el zoom del mapa. Las acciones se crean para la activación de cada herramienta: la panorámica se realiza con `QgsMapToolPan`, acercando/alejando con un par de instancias `QgsMapToolZoom`. Las acciones se configuran como verificables y luego se asignan a las herramientas para permitir el manejo automático del estado verificado / no verificado de las acciones: cuando una herramienta de mapa se activa, su acción se marca como seleccionada y la acción de la herramienta de mapa anterior se deselecciona. Las herramientas de mapa se activan usando `setMapTool()` method.

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

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

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

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

        self.setCentralWidget(self.canvas)

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

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

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

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

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

        self.pan()

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

    def zoomOut(self):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
        self.canvas.setMapTool(self.toolZoomOut)

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

Puede probar el código anterior en el editor de la consola de Python. Para invocar la ventana del lienzo, agregue las siguientes líneas para crear una instancia de la clase ``MyWnd"". Representarán la capa seleccionada actualmente en el lienzo recién creado

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

9.3.1 Selecciona una entidad usando QgsMapToolIdentifyFeature

Puedes usar la herramienta del mapa `QgsMapToolIdentifyFeature` para pedirle al usuario que seleccione una entidad que se enviará a una función de devolución de llamada.

```
def callback(feature):
  """Code called when the feature is selected by the user"""
  print("You clicked on feature {}".format(feature.id()))

canvas = iface.mapCanvas()
feature_identifier = QgsMapToolIdentifyFeature(canvas)

# indicates the layer on which the selection will be done
feature_identifier.setLayer(vlayer)

# use the callback as a slot triggered when the user identifies a feature
feature_identifier.featureIdentified.connect(callback)

# activation of the map tool
canvas.setMapTool(feature_identifier)
```

9.3.2 Añadir temas al menú contextual del lienzo del mapa

La interacción con el lienzo del mapa se puede hacer también por medio de entradas que usted puede hacer en su menú contextual usando la señal `contextMenuAboutToShow`.

El siguiente código añade la acción *Mi menú* ► *Mi Acción* cerca de las entradas predeterminadas cuando hace pulsación derecha sobre el lienzo del mapa.

```
# a slot to populate the context menu
def populateContextMenu(menu: QMenu, event: QgsMapMouseEvent):
    subMenu = menu.addMenu('My Menu')
    action = subMenu.addAction('My Action')
    action.triggered.connect(lambda *args:
                             print(f'Action triggered at {event.x()},{event.y()}'))

canvas.contextMenuAboutToShow.connect(populateContextMenu)
canvas.show()
```

9.4 Escribir herramientas de mapa personalizados

Puede escribir sus herramientas personalizadas, para implementar un comportamiento personalizado a las acciones realizadas por los usuarios en el lienzo.

Las herramientas de mapa deben heredar de `QgsMapTool`, clase o cualquier clase derivada, y seleccionada como herramientas activas en el lienzo utilizando el método `setMapTool()` como ya hemos visto.

Aquí esta un ejemplo de una herramienta de mapa para definir una extensión rectangular haciendo clic y arrastrando en el lienzo. Cuando se define el rectángulo, imprime su limite de coordenadas en la consola. Utiliza los elementos de la banda elástica descrita antes para mostrar el rectángulo seleccionado ya que se esta definiendo.

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

  def reset(self):
    self.startPoint = self.endPoint = None
    self.isEmittingPoint = False
    self.rubberBand.reset(QgsWkbTypes.PolygonGeometry)

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

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

  def canvasMoveEvent(self, e):
    if not self.isEmittingPoint:
      return

    self.endPoint = self.toMapCoordinates(e.pos())
    self.showRect(self.startPoint, self.endPoint)

  def showRect(self, startPoint, endPoint):
    self.rubberBand.reset(QgsWkbTypes.PolygonGeometry)
    if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
      return

    point1 = QgsPointXY(startPoint.x(), startPoint.y())
    point2 = QgsPointXY(startPoint.x(), endPoint.y())
    point3 = QgsPointXY(endPoint.x(), endPoint.y())
    point4 = QgsPointXY(endPoint.x(), startPoint.y())

    self.rubberBand.addPoint(point1, False)
    self.rubberBand.addPoint(point2, False)
    self.rubberBand.addPoint(point3, False)
    self.rubberBand.addPoint(point4, True)    # true to update canvas
    self.rubberBand.show()
```

(continúe en la próxima página)

(proviene de la página anterior)

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

      return QgsRectangle(self.startPoint, self.endPoint)

  def deactivate(self):
    QgsMapTool.deactivate(self)
    self.deactivated.emit()
```

9.5 Escribir elementos de lienzo de mapa personalizado

Aquí hay un ejemplo de un elemento de lienzo personalizado que dibuja un círculo:

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

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

  def center(self):
    return self.center

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

  def size(self):
    return self.size

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

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


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

CAPÍTULO 10

Representación del Mapa e Impresión

Consejo: Los fragmentos de código en esta página necesitan las siguientes importaciones:

```
import os

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

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

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

Por lo general, hay dos enfoques cuando los datos de entrada deben representarse como un mapa: hacerlo de manera rápida usando *QgsMapRendererJob* o producir una salida más ajustada componiendo el mapa con la clase `QgsLayout`.

10.1 Representación Simple

La renderización se realiza creando un objeto `QgsMapSettings` para definir la configuración de renderizado, y luego construir una clase `QgsMapRendererJob` con esos ajustes. Este último se utiliza para crear la imagen resultante.

He aquí un ejemplo:

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

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

render = QgsMapRendererParallelJob(settings)

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

render.finished.connect(finished)

# Start the rendering
render.start()

# The following loop is not normally required, we
# are using it here because this is a standalone example.
from qgis.PyQt.QtCore import QEventLoop
loop = QEventLoop()
render.finished.connect(loop.quit)
loop.exec_()
```

10.2 Representando capas con diferente SRC

Si tiene más de una capa y tienen un SRC diferente, el simple ejemplo anterior probablemente no funcionará: para obtener los valores correctos de los cálculos de extensión, debe establecer explícitamente el SRC de destino

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

10.3 Salida usando diseño de impresión

El diseño de impresión es una herramienta muy útil si desea realizar una salida más sofisticada que la simple representación mostrada anteriormente. Es posible crear diseños de mapas complejos compuestos por vistas de mapas, etiquetas, leyendas, tablas y otros elementos que suelen estar presentes en los mapas en papel. Los diseños pueden exportarse a PDF, SVG, imágenes rasterizadas o imprimirse directamente en una impresora.

El diseño consta de varias clases. Todos pertenecen a la biblioteca principal. La aplicación QGIS tiene una GUI conveniente para la ubicación de los elementos, aunque no está disponible en la biblioteca de la GUI. Si no está familiarizado con Qt Graphics View framework, entonces le recomendamos que consulte la documentación ahora, porque el diseño se basa en ella.

La clase central del diseño es la clase `QgsLayout`, que deriva de la clase de Qt QGraphicsScene . Creemos una instancia de ello:

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

Esto inicializa el diseño con algunas configuraciones predeterminadas, específicamente agregando una página A4 vacía al diseño. Puede crear diseños sin llamar al método `initializeDefaults()`, pero deberá encargarse de agregar páginas al diseño usted mismo.

El código anterior crea un diseño «temporal» que no es visible en la GUI. Puede ser útil p.Ej. agregue rápidamente algunos elementos y exporte sin modificar el proyecto en sí ni exponer estos cambios al usuario. Si desea que el diseño se guarde/restaure junto con el proyecto y esté disponible en el administrador de diseño, agregue:

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

Ahora podemos agregar varios elementos (mapa, etiqueta, …) al diseño. Todos estos objetos están representados por clases que heredan de la clase base `QgsLayoutItem`.

Aquí hay una descripción de algunos de los elementos de diseño principales que se pueden agregar a un diseño.

- mapa — Aquí creamos un mapa de un tamaño personalizado y renderizamos el lienzo del mapa actual.

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

- etiqueta — permite mostrar etiquetas. Es posible modificar su letra, color, alineación y margen.

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

- legend

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

- barra de escala

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

- forma basada en nodos

```
polygon = QPolygonF()
polygon.append(QPointF(0.0, 0.0))
polygon.append(QPointF(100.0, 0.0))
polygon.append(QPointF(200.0, 100.0))
polygon.append(QPointF(100.0, 200.0))

polygonItem = QgsLayoutItemPolygon(polygon, layout)
layout.addLayoutItem(polygonItem)

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

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

Una vez que un elemento es añadido a la composiciónm puede ser movida y redimensionadas:

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

Un cuadro es dibujado alrededor de cada elemento por defecto. Puede borrarlo como sigue:

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

Además de crear los elementos de diseño a mano, QGIS tiene soporte para plantillas de diseño que son esencialmente composiciones con todos sus elementos guardados en un archivo .qpt (con sintaxis XML).

Una vez que la composición está lista (los elementos de diseño se han creado y agregado a la composición), podemos proceder a producir una salida rasterizada y/o vectorial.

10.3.1 Comprobación de la validez del diseño

Un diseño está formado por un conjunto de elementos interconectados y puede ocurrir que estas conexiones se rompan durante las modificaciones (una leyenda conectada a un mapa eliminado, un elemento de imagen al que le falta el archivo fuente,…) o puede que desee aplicar restricciones personalizadas a los elementos del diseño. La `QgsAbstractValidityCheck` te ayuda a conseguirlo.

Una comprobación básica tiene este aspecto:

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

Aquí hay una comprobación que lanza una advertencia cada vez que un elemento de mapa de diseño se establece en la proyección mercator web:

```
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def layout_map_crs_choice_check(context, feedback):
  layout = context.layout
  results = []
  for i in layout.items():
    if isinstance(i, QgsLayoutItemMap) and i.crs().authid() == 'EPSG:3857':
      res = QgsValidityCheckResult()
      res.type = QgsValidityCheckResult.Warning
      res.title = 'Map projection is misleading'
      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())
      results.append(res)

  return results
```

Y aquí hay un ejemplo más complejo, que lanza una advertencia si cualquier elemento de mapa de diseño se establece en un SRC que sólo es válido fuera de la extensión mostrada en ese elemento de mapa:

```
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def layout_map_crs_area_check(context, feedback):
    layout = context.layout
    results = []
    for i in layout.items():
        if isinstance(i, QgsLayoutItemMap):
            bounds = i.crs().bounds()
            ct = QgsCoordinateTransform(QgsCoordinateReferenceSystem('EPSG:4326'),␣
↪i.crs(), QgsProject.instance())
            bounds_crs = ct.transformBoundingBox(bounds)

            if not bounds_crs.contains(i.extent()):
                res = QgsValidityCheckResult()
                res.type = QgsValidityCheckResult.Warning
                res.title = 'Map projection is incorrect'
                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())
                results.append(res)

    return results
```

10.3.2 Exportando la composición

Para exportar una composición, la clase `QgsLayoutExporter` debe ser usada.

```
base_path = os.path.join(QgsProject.instance().homePath())
pdf_path = os.path.join(base_path, "output.pdf")

exporter = QgsLayoutExporter(layout)
exporter.exportToPdf(pdf_path, QgsLayoutExporter.PdfExportSettings())
```

Utilice `exportToSvg()` o `exportToImage()` en caso de que desee exportar respectivamente a un archivo SVG o de imagen en lugar de a un archivo PDF.

10.3.3 Exportar un atlas de diseño

Si desea exportar todas las páginas de un diseño que tiene la opción de atlas configurada y habilitada, debe usar el método `atlas()` en el exportador (`QgsLayoutExporter`) con pequeños ajustes. En el siguiente ejemplo, las páginas se exportan a imágenes PNG:

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

Observe que las salidas se guardarán en la carpeta de ruta base, utilizando la expresión de nombre de archivo de salida configurada en atlas.

CAPÍTULO 11

Expresiones, Filtros y Calculando Valores

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

QGIS tiene cierto soporte para analizar expresiones similares a SQL. Solo se admite un pequeño subconjunto de sintaxis SQL. Las expresiones se pueden evaluar como predicados booleanos (devolviendo `True` o `False`) o como funciones (devolviendo un valor escalar). Consulte vector_expressions en el Manual del usuario para obtener una lista completa de las funciones disponibles.

Se le da apoyo a tres tipos:

- numero - números enteros y números con decimales, e.j. `123`, `3.14`
- cadena - se tiene que encerrar en comas individuales: `'hola mundo'`
- columna de referencia - cuando se evalúa, la referencia se substituye con el valor actual del campo. Los nombres no se escapan.

Los siguientes operadores están disponibles:

- operadores aritméticos: «+». «-», «/», `^`
- paréntesis: para hacer cumplir la precedencia del operador: `(1 + 1) * 3`
- unario mas y menos: `-12`, `+5`

- funciones matemáticas: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- funciones de conversión: `to_int`, `to_real`, `to_string`, `to_date`
- funciones geométricas: `$area`, `$length`
- funciones de manejo de geometría: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

Se apoya las siguientes predicciones:

- comparación: =, `!=`, >, >=, <, <=
- patrones iguales: `LIKE` (using % and _), ~ (expresión regular)
- lógica predicado: `AND`, `OR`, `NOT`
- revisión de valores NULO: `IS NULL`, `IS NOT NULL`

Ejemplos de predicado:

- `1 + 2 = 3`
- `sin(angulo) > 0`
- *"Hello" LIKE "He%"*
- `(x > 10 AND y > 10) OR z = 0`

Ejemplo de escala de expresiones:

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

11.1 Análisis de expresiones

El siguiente ejemplo muestra cómo verificar si una expresión dada se puede analizar correctamente:

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

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

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

11.2 Evaluar expresiones

Las expresiones se pueden usar en diferentes contextos, por ejemplo, para filtrar entidades o para calcular nuevos valores de campo. En cualquier caso, la expresión tiene que ser evaluada. Eso significa que su valor se calcula realizando los pasos computacionales especificados, que pueden variar desde expresiones aritméticas simples hasta expresiones agregadas.

11.2.1 Expresiones Basicas

Esta expresión básica evalúa una operación aritmética simple:

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

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

La expresión también se puede utilizar para comparar, evaluando 1 (`True`) o 0 (`False`)

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

11.2.2 Expresiones con caracteristicas

Para evaluar una expresión sobre una entidad, se debe crear un objeto **:clase:`QgsExpressionContext <qgis.core.QgsExpressionContext>`** y pasarlo a la función de evaluación para permitir que la expresión acceda a los valores de campo de la entidad.

El siguiente ejemplo muestra cómo crear una entidad con un campo llamado «Columna» y cómo agregar esta entidad al contexto de expresión.

```
fields = QgsFields()
field = QgsField('Column')
fields.append(field)
feature = QgsFeature()
feature.setFields(fields)
feature.setAttribute(0, 99)

exp = QgsExpression('"Column"')
context = QgsExpressionContext()
context.setFeature(feature)
exp.evaluate(context)
# 99
```

El siguiente es un ejemplo más completo de cómo usar expresiones en el contexto de una capa vectorial, para calcular nuevos valores de campo:

```
from qgis.PyQt.QtCore import QVariant

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```

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

vl.updateExtents()
QgsProject.instance().addMapLayer(vl)

# The first expression computes the revenue per employee.
# The second one computes the sum of all revenue values in the layer.
# The final third expression doesn't really make sense but illustrates
# the fact that we can use a wide range of expression functions, such
# as area and buffer in our expressions:
expression1 = QgsExpression('"Revenue"/"Employees"')
expression2 = QgsExpression('sum("Revenue")')
expression3 = QgsExpression('area(buffer($geometry,"Employees"))')

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

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

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

```
876.5
```

11.2.3 Flitrando una capa con expresiones

El siguiente ejemplo se puede utilizar para filtra capas y regresar cualquier característica que empata con el predicado.

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

layer.startEditing()

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

expression = 'Test >= 3'
request = QgsFeatureRequest().setFilterExpression(expression)

matches = 0
```

(continúe en la próxima página)

(proviene de la página anterior)

```
for f in layer.getFeatures(request):
   matches += 1

print(matches)
```

```
7
```

11.3 Manejando errores de expresión

Los errores relacionados con la expresión pueden ocurrir durante el análisis o la evaluación de expresiones:

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

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

CAPÍTULO 12

Configuración de lectura y almacenamiento

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

Muchas veces es útil para un complemento guardar algunas variables para que el usuario no tenga que introducir o seleccionar de nuevo la próxima vez que el complemento se ejecute.

Estas variables se pueden guardar y recuperar con ayuda de Qt y QGIS API. Para cada variable, se debe escoger una clave que será utilizada para acceder a la variable — para el color favorito del usuario podría utilizarse la clave «favourite_color» o cualquier otra cadena que tenga sentido. Es recomendable dar un poco de estructura al nombrar las claves.

Podemos diferenciar entre varios tipos de configuraciones:

- **global settings** — están vinculados al usuario en una máquina en particular. QGIS almacena muchas configuraciones globales, por ejemplo, el tamaño de la ventana principal o la tolerancia de ajuste predeterminada. La configuración se maneja usando la clase `QgsSettings`, por ejemplo, a través de `setValue()` y :meth:`value() < qgis.core.QgsSettings.value>`métodos.

  Aquí puede ver un ejemmplo de como se usan estos métodos.

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

def read():
  s = QgsSettings()
  mytext = s.value("myplugin/mytext", "default text")
  myint  = s.value("myplugin/myint", 123)
  myreal = s.value("myplugin/myreal", 2.71)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  nonexistent = s.value("myplugin/nonexistent", None)
  print(mytext)
  print(myint)
  print(myreal)
  print(nonexistent)
```

El segundo parámetro del método `value()` es opcional y especifica el valor predeterminado que se devuelve si no hay un valor anterior establecido para el nombre de configuración pasado.

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.

- **project settings** — varían entre diferentes proyectos y, por lo tanto, están conectados con un archivo de proyecto. El color de fondo del lienzo del mapa o el sistema de referencia de coordenadas de destino (CRS) son ejemplos: el fondo blanco y WGS84 pueden ser adecuados para un proyecto, mientras que el fondo amarillo y la proyección UTM son mejores para otro.

  A continuación se muestra un ejemplo de uso.

```
proj = QgsProject.instance()

# store values
proj.writeEntry("myplugin", "mytext", "hello world")
proj.writeEntry("myplugin", "myint", 10)
proj.writeEntryDouble("myplugin", "mydouble", 0.01)
proj.writeEntryBool("myplugin", "mybool", True)

# read values (returns a tuple with the value, and a status boolean
# which communicates whether the value retrieved could be converted to
# its type, in these cases a string, an integer, a double and a boolean
# respectively)

mytext, type_conversion_ok = proj.readEntry("myplugin",
                                            "mytext",
                                            "default text")
myint, type_conversion_ok = proj.readNumEntry("myplugin",
                                              "myint",
                                              123)
mydouble, type_conversion_ok = proj.readDoubleEntry("myplugin",
                                                    "mydouble",
                                                    123)
mybool, type_conversion_ok = proj.readBoolEntry("myplugin",
                                                "mybool",
                                                123)
```

Como puede ver, el método `writeEntry()` es usado para muchos tipos de datos (entero, cadena, lista), pero existen varios métodos para volver a leer el valor de ajuste, y se debe seleccionar el correspondiente para cada tipo de datos.

- **map layer settings** — esta configuración está relacionada con una instancia particular de una capa de mapa con un proyecto. No están *conectados* con la fuente de datos subyacente de una capa, por lo que si crea dos instancias de capa de mapa de un archivo de forma, no compartirán la configuración. La configuración se almacena dentro del archivo del proyecto, por lo que si el usuario abre el proyecto nuevamente, la configuración relacionada con la capa volverá a estar allí. El valor para una configuración dada se recupera usando el método `customProperty()`, y se puede establecer usando el método `setCustomProperty()` uno.

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

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

CAPÍTULO 13

Comunicarse con el usuario

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

from qgis.gui import (
    QgsMessageBar,
)

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

Esta sección muestra algunos métodos y elementos que deberían utilizarse para comunicarse con el usuario, con el objetivo de mantener la consistencia en la Interfaz del Usuario.

13.1 Mostrando mensajes. La clase QgsMessageBar

Utilizar las bandejas de mensajes puede ser una mala idea desde el punto de vista de la experiencia de un usuario. Para mostrar una pequeña línea de información o mensajes de advertencia/error, la barrar de mensajes de QGIS suele ser una mejor opción.

Utilizar la referencia a la interfaz objeto de QGIS, puede mostrar un mensaje en la barra de mensajes con el siguiente código

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

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

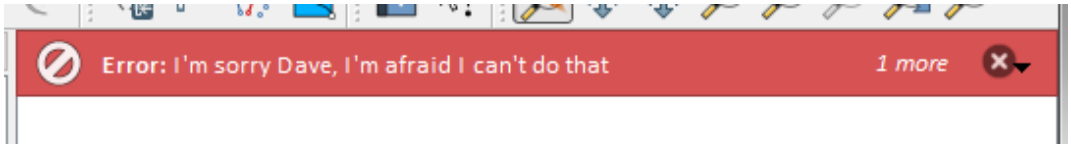


Figura 13.1: Barra de Mensajes de QGIS

Se puede establecer una duración para mostrarlo en un tiempo limitado

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

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

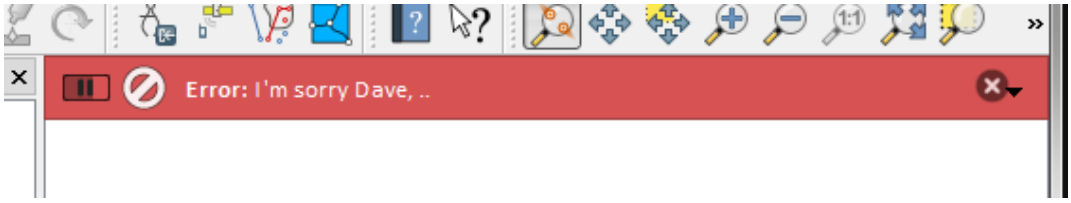


Figura 13.2: Barra de Mensajes de QGIS con temporizador

Los ejemplos anteriores muestran una barra de error, pero el parámetro `level` se puede usar para crear mensajes de advertencia o mensajes de información, utilizando la enumeración `Qgis.MessageLevel`. Puede usar hasta 4 niveles diferentes:

0. Información
1. Advertencia
2. Critica
3. Éxito

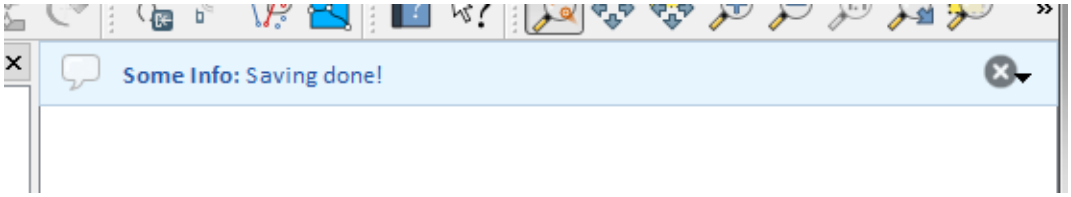


Figura 13.3: Barra de Mensajes de QGIS (información)

Se puede añadir complementos a la barra de mensajes, como por ejemplo un botón para mostrar más información

```
def showError():
    pass

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

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

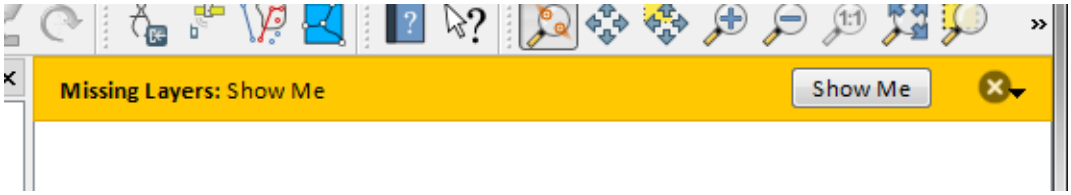


Figura 13.4: Barra de Mensajes de QGIS con un botón

Incluso puedes utilizar una barra de mensajes en tu propio cuadro de diálogo para no tener que mostrar la bandeja de mensajes o si no tiene sentido mostrarla en la pantalla principal de QGIS.

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

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

Figura 13.5: Barra de Mensajes de QGIS con un cuadro de diálogo personalizado

13.2 Mostrando el progreso

Las barras de progreso también pueden ponerse en la barra de Mensajes de QGIS, ya que, como hemos visto, admite complementos. Este es un ejemplo que puedes probar en la consola.

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

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

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

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

Además, puede usar el constructor en la barra de estado para comunicar progreso, como en el siguiente ejemplo:

```
vlayer = iface.activeLayer()

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

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

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

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

13.3 Registro

Hay tres tipos diferentes de registro disponible en QGIS para registrar y guardar toda la información sobre la ejecución de su código. Cada uno tiene una ubicación de salida específica. Por favor considere el uso de la forma correcta de registro para su propósito:

- `QgsMessageLog` es para que los mensajes comuniquen problemas al usuario. La salida del QgsMessageLog es mostrada en el Panel de Mensajes de Registro.
- El python construido en el módilo **registro** es para depuración en el nivel de la API Python QGIS (PyQGIS). Es recomendado para desarrolladores de script Python que necesitan depurar su código python, e.g. ids o geometría de objeto espacial
- `QgsLogger` es para mensajes para depuración / desarrolladores *internos QGIS* (i.e. usted sospecha que algo es gatillado por algún código dañado). Los mensajes sólo están visibles con versiones desarrollador de QGIS.

Ejemplos para los diferentes tipos de registro son mostrados en las siguientes secciones abajo.

Advertencia: El uso de la declaración `print` Python no es segura en ningún código que sea multihilos y **ralentiza extremadamente el algoritmo**. Esto incluye **funciones de expresión**, **representadores**, **capas de símbolo** y **algoritmos Procesos** (entre otros). En estos casos usted siempre debería usar en vez el módulo python **registro** o clases seguras de hilo (`QgsLogger` o `QgsMessageLog`).

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: Puede ver la salida de la `QgsMessageLog` en el log_message_panel

13.3.2 El python construído en el módulo de registro

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

El método basicConfig configura la definición básica del registro. En el código de arriba el nombre de archivo, nivel de registro y el formato son definidos. El nombre de archivo se refiere a donde escribir el archivo de registro, el nivel de registro define qué niveles imprimir y el formato define el formato en el que cada mensaje es impreso.

```
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
```

Si quiere borrar el archivo de registro cada vez que ejecuta su script usted puede hacer algo como:

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

Mayores recursos sobre cómo usar la instalación de registro python están disponibles en:

- 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

Advertencia: Por favor tome nota que sin registrar a un archivo al definir el nombre de archivo, el registro podría darse en multihilos lo que ralentiza fuertemente la salida.

CAPÍTULO 14

Infraestructura de autenticación

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

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

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

from qgis.PyQt.QtNetwork import QSslCertificate
```

14.1 Introducción

La referencia de Usuario de la infraestructura de Autenticación puede ser leída en el Manual del Usuario en el párrafo authentication_overview.

En este capítulo se describen las mejores prácticas para usar el sistema de autenticación desde una perspectiva de desarrollador.

El sistema de autenticación es ampliamente utilizado en QGIS Desktop por los proveedores de datos siempre que se requieren credenciales para acceder a un recurso en particular, por ejemplo, cuando una capa establece una conexión a una base de datos de Postgres.

También hay algunos widgets en la biblioteca de interfaz gráfica de usuario de QGIS que los desarrolladores de complementos pueden usar para integrar fácilmente la infraestructura de autenticación en su código:

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

Se puede leer una buena referencia de código desde la infraestructura de autenticación tests code.

Advertencia: Debido a las restricciones de seguridad que se tuvieron en cuenta durante el diseño de la infraestructura de autenticación, solo un subconjunto seleccionado de los métodos internos están expuestos a Python.

14.2 Glosario

Aquí hay algunas definiciones de los objetos más comunes tratados en este capítulo.

Contraseña maestra
: Contraseña para permitir el acceso y descifrar las credenciales almacenadas en la base de datos de autenticación de QGIS

BBDD de Autenticación
: Un *Master Password* encriptado sqlite db `qgis-auth.db` donde *Authentication Configuration* son almacenados. p.ej. usuario/contraseña, certificados personales y claves, Autoridades de certificación

Autenticación BD
: *Authentication Database*

Configuración de Autenticación
: Un conjunto de datos de autenticación según *Authentication Method*. p.ej. El método de Autenticación básica almacena el par de usuario/contraseña.

Configuración de autenticación
: *Authentication Configuration*

Método de autenticación
: Un método específico utilizado para autenticarse. Cada método tiene su propio protocolo utilizado para obtener el nivel autenticado. Cada método se implementa como una biblioteca compartida cargada dinámicamente durante el inicio de la infraestructura de autenticación de QGIS.

14.3 QgsAuthManager el punto de entrada

El `QgsAuthManager` singleton es el punto de entrada para usar las credenciales almacenadas en el QGIS encriptado BBDD Autenticación, es decir, el archivo `qgis-auth.db` en la carpeta activa perfil de usuario.

Esta clase se encarga de la interacción del usuario: solicitando establecer una contraseña maestra o utilizándola de forma transparente para acceder a la información almacenada cifrada.

14.3.1 Inicie el administrador y configure la contraseña maestra

El siguiente fragmento ofrece un ejemplo para establecer una contraseña maestra para abrir el acceso a la configuración de autenticación. Los comentarios de código son importantes para comprender el fragmento.

```
authMgr = QgsApplication.authManager()

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

14.3.2 Complete authdb con una nueva entrada de configuración de autenticación

Cualquier credencial almacenado es una instancia *Authentication Configuration* de la clase `QgsAuthMethodConfig` se accede usando una cadena única como la siguiente:

```
authcfg = 'fm1s770'
```

esa cadena se genera automáticamente al crear una entrada utilizando la API o GUI de QGIS, pero podría ser útil establecerla manualmente en un valor conocido en caso de que la configuración deba compartirse (con diferentes credenciales) entre varios usuarios dentro de una organización.

`QgsAuthMethodConfig` es la clase base para cualquier *Método de autenticación*. Cualquier método de autenticación establece un mapa hash de configuración donde se almacenará la información de autenticación. A continuación, un fragmento útil para almacenar las credenciales de ruta PKI para un usuario hipotético de Alice:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

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

Métodos de autenticación disponibles

Authentication Method las bibliotecas se cargan dinámicamente durante el inicio del administrador de autenticación. Los métodos de autenticación disponibles son:

1. `Basic` Autenticación de usuario y contraseña
2. `EsriToken` ESRI token based authentication
3. `Identity-Cert` Autenticación del certificado de identidad
4. `OAuth2` autenticación OAuth2
5. `PKI-Paths` Autenticación de rutas PKI
6. `PKI-PKCS#12` PKI PKCS#12 Autenticación

Populate Authorities

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

Administrar paquetes de PKI con QgsPkiBundle

Una clase de conveniencia para empaquetar paquetes PKI compuestos en cadenas SslCert, SslKey y CA es la clase `QgsPkiBundle`. A continuación, un fragmento para protegerse con contraseña:

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

Referirse a documentación de la clase `QgsPkiBundle` para extraer cert/clave/CAs del paquete.

14.3.3 Borrar una entrada de authdb

Podemos borrar una entrada de *Authentication Database* usando su identificador `authcfg` con el siguiente fragmento:

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

14.3.4 Deje la expansión authcfg a QgsAuthManager

La mejor manera de utilizar una *Configuración de Autenticación* almacenada en *BBDD de Autenticación* es refiriéndola con el identificador único `authcfg`. Expandir significa convertirlo de un identificador a un conjunto completo de credenciales. La mejor práctica para usar el *Authentication Config*s almacenado, es dejarlo administrado automáticamente por el administrador de Autenticación. El uso común de una configuración almacenada es conectarse a un servicio habilitado para autenticación como WMS o WFS oa una conexión de base de datos.

Nota: Tenga en cuenta que no todos los proveedores de datos de QGIS están integrados con la infraestructura de autenticación. Cada método de autenticación, derivado de la clase base `QgsAuthMethod` y soporta un conjunto diferente de Proveedores. Por ejemplo el método `certIdentity()` soporta la siguiente lista de proveedores:

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

Salida de muestra:

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

Por ejemplo, para acceder a un servicio WMS usando credenciales almacenadas identificadas con `authcfg = 'fm1s770'`, solo tenemos que usar el `authcfg` en la URL de la fuente de datos como en el siguiente fragmento:

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

En mayúsculas, el proveedor `wms` se encargará de expandir el parámetro URI `authcfg` con credencial justo antes de configurar la conexión HTTP.

Advertencia: El desarrollador tendría que dejar la expansión *`authcfg`* a `QgsAuthManager`, de esta manera se asegurará de que la expansión no se haga demasiado pronto.

Por lo general, una cadena URI, construida usando la clase `QgsDataSourceURI` , se utiliza para configurar una fuente de datos de la siguiente manera:

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

Nota: El parámetro `False` es importante para evitar la expansión completa de URI del id `authcfg` presente en el URI.

Ejemplos PKI con otros proveedores de datos

Otro ejemplo se puede leer directamente en las pruebas de QGIS en sentido ascendente como en test_authmanager_pki_ows o test_authmanager_pki_postgres.

14.4 Adpatar complementos para usar Infraestructura de Autenticación

Muchos complementos de terceros están utilizando httplib2 u otras bibliotecas de red de Python para administrar las conexiones HTTP en lugar de integrarse con `QgsNetworkAccessManager` y su integración relacionada con la infraestructura de autenticación.

Para facilitar esta integración se ha creado una función auxiliar de Python llamada `NetworkAccessManager`. Su código se puede encontrar aquí.

Esta clase auxiliar se puede utilizar como en el siguiente fragmento:

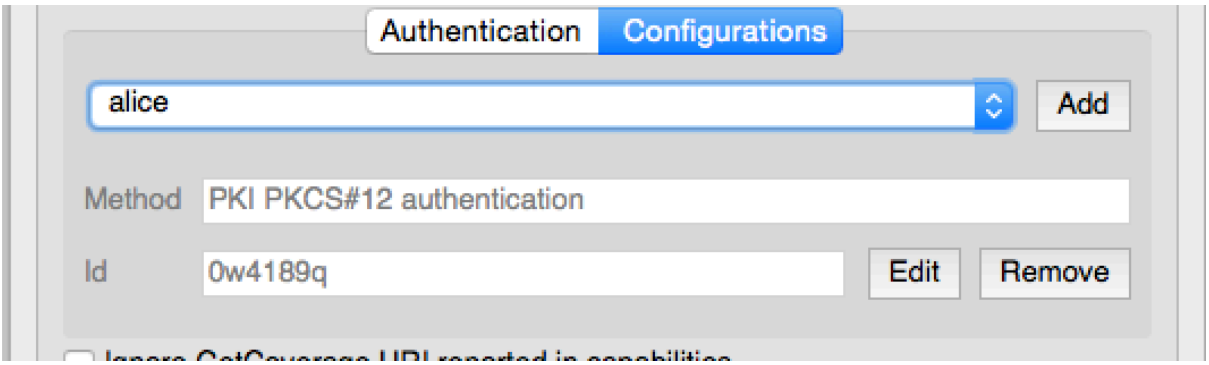
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 Autenticación IGUs

En este párrafo se enumeran las IGU disponibles útiles para integrar la infraestructura de autenticación en interfaces personalizadas.

14.5.1 IGU para seleccionar credenciales

Si es necesario seleccionar un *Authentication Configuration* del conjunto almacenado en la BBDD Autenticación está disponible en la clase IGU `QgsAuthConfigSelect`.



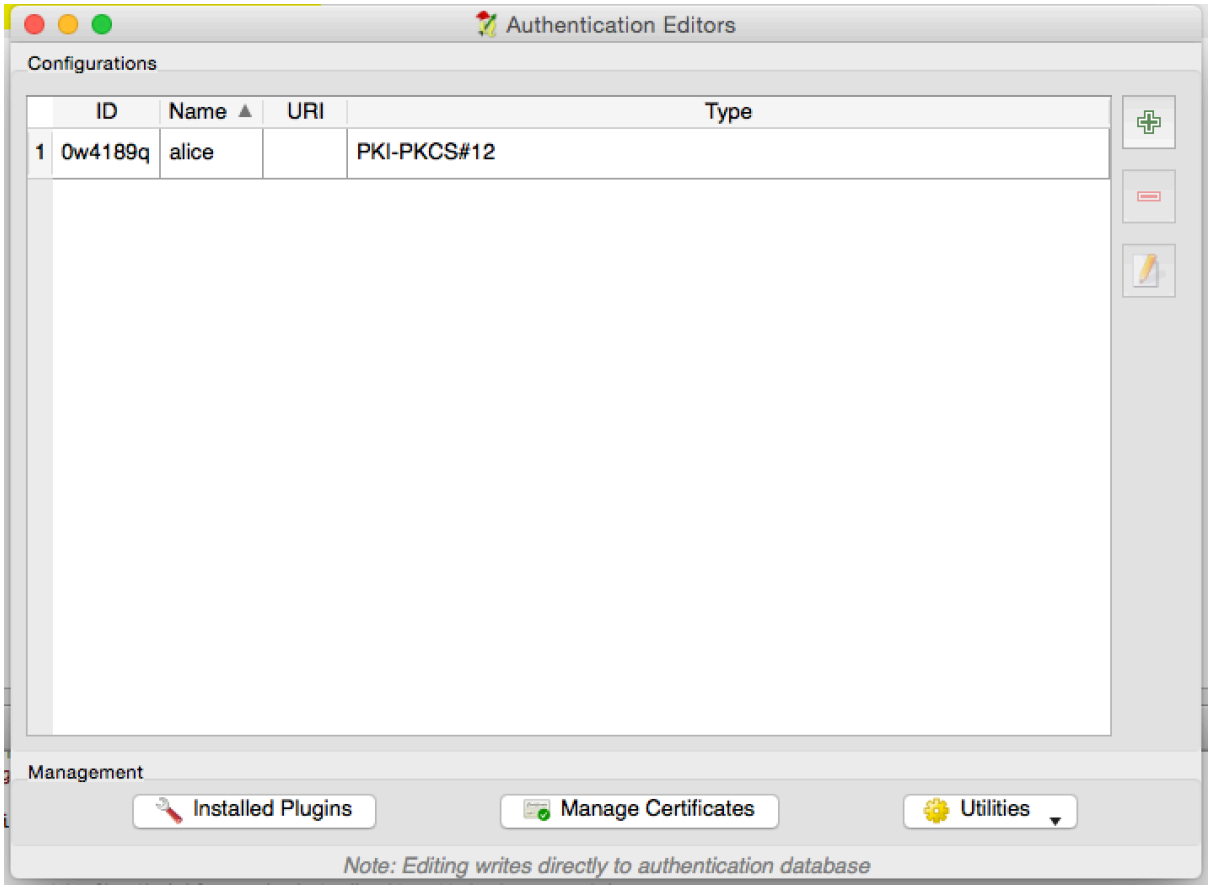
y se puede utilizar como en el siguiente fragmento:

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

El ejemplo anterior está tomado de la fuente QGIS code. El segundo parámetro del constructor de GUI se refiere al tipo de proveedor de datos. El parámetro se utiliza para restringir el *Método de autenticación* compatible con el proveedor especificado.

14.5.2 IGU Editor Autenticación

La GUI completa utilizada para administrar credenciales, autoridades y para acceder a las utilidades de autenticación es administrada por la clase `QgsAuthEditorWidgets`.



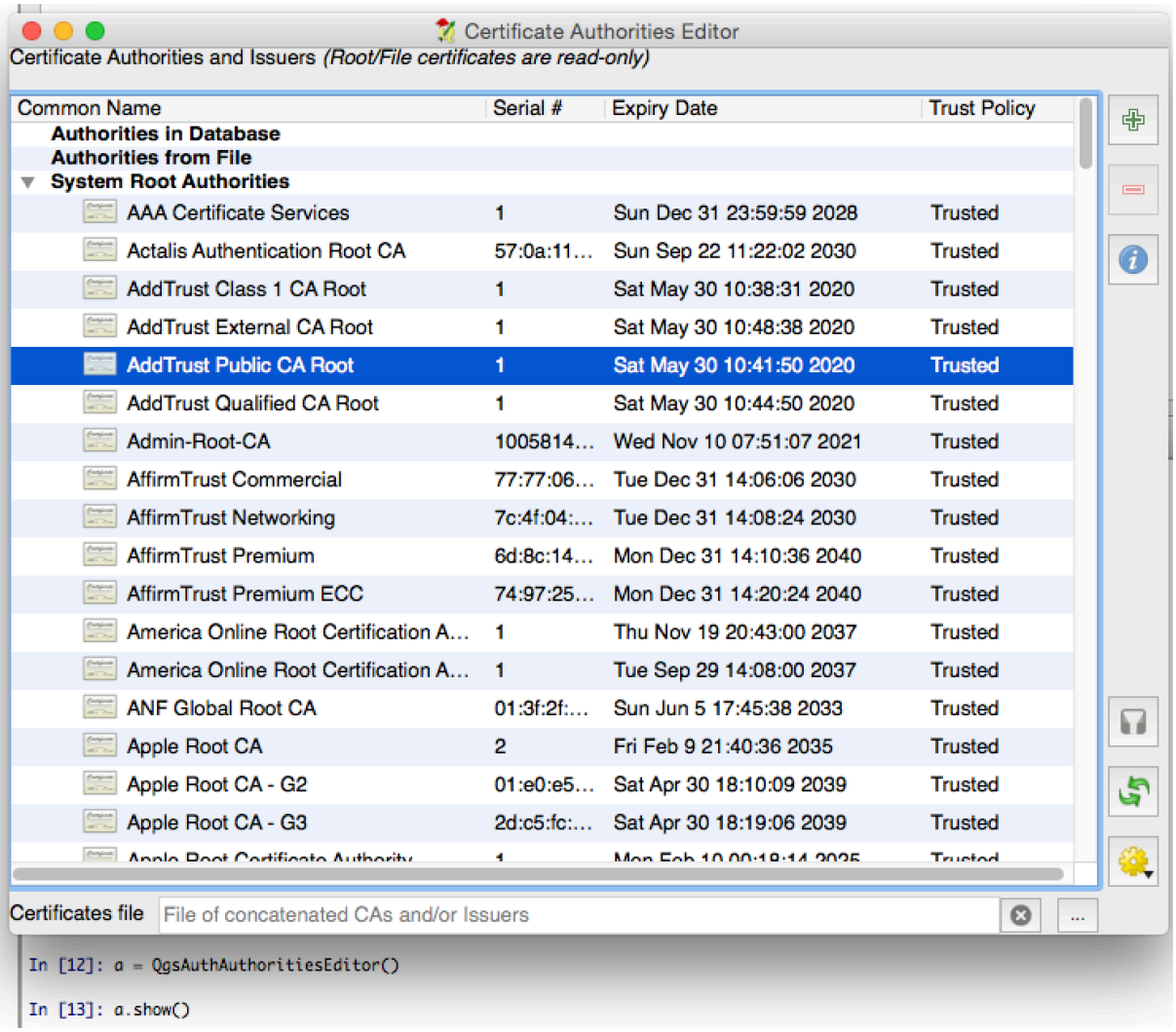
y se puede utilizar como en el siguiente fragmento:

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

Se puede encontrar un ejemplo integrado en el test.

14.5.3 IGU de Editor de Autoridades

Una GUI utilizada para administrar solo las autoridades es administrada por la clase `QgsAuthAuthoritiesEditor`.



y se puede utilizar como en el siguiente fragmento:

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

CAPÍTULO 15

Tareas - haciendo trabajo duro en segundo plano

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

15.1 Introducción

El procesamiento en segundo plano utilizando subprocesos es una forma de mantener una interfaz de usuario receptiva cuando se está realizando un procesamiento pesado. Las tareas se pueden utilizar para realizar subprocesos en QGIS.

Una tarea (`QgsTask`) es un contenedor para el código que se realizará en segundo plano, y el administrador de tareas (`QgsTaskManager`) se usa para controlas la ejecución de las tareas. Estas clases simplifican el procesamiento en segundo plano en QGIS al proporcionar mecanismos de señalización, informes de progreso y acceso al estado de los procesos en segundo plano. Las tareas se pueden agrupar mediante subtareas.

El administrador de tareas global (encontrado con `QgsApplication.taskManager()`) es usado normalmente. Esto significa que sus tareas pueden no ser las únicas tareas controladas por el administrador de tareas.

Hay varias vías para crear una tarea QGIS:

- Crea tu propia tarea extendiendo `QgsTask`

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

- Crear una tarea desde una función

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

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

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

- Crear una tarea desde un algoritmo de procesamiento

```
params = dict()
context = QgsProcessingContext()
context.setProject(QgsProject.instance())
feedback = QgsProcessingFeedback()

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

Advertencia: Cualquier tarea en segundo plano (independientemente de cómo se cree) NUNCA debe usar ningún QObject que viva en el hilo principal, como acceder a QgsVectorLayer, QgsProject o realizar cualquier operación basada en GUI como crear nuevos widgets o interactuar con widgets existentes. Solo se debe acceder o modificar los widgets de Qt desde el hilo principal. Los datos que se utilizan en una tarea se deben copiar antes de iniciar la tarea. Intentar utilizarlos desde subprocesos en segundo plano provocará bloqueos.

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.

Las dependencias entre tareas se pueden describir utilizando la función `addSubTask()` de `QgsTask`. Cuando se establece una dependencia, el administrador de tareas determinará automáticamente cómo se ejecutarán estas dependencias. Siempre que sea posible, las dependencias se ejecutarán en paralelo para satisfacerlas lo más rápido posible. Si se cancela una tarea de la que depende otra tarea, la tarea dependiente también se cancelará. Las dependencias circulares pueden hacer posibles los interbloqueos, así que tenga cuidado.

Si una tarea depende de que haya una capa disponible, esto se puede indicar usando la función `setDependentLayers()` de `QgsTask`. Si una capa de la que depende una tarea no está disponible, la tarea se cancelará.

Una vez que se ha creado la tarea, se puede programar su ejecución utilizando la función `addTask()` del administrador de tareas. Agregar una tarea al administrador transfiere automáticamente la propiedad de esa tarea al administrador, y el administrador limpiará y eliminará las tareas después de que se hayan ejecutado. La programación de las tareas está influenciada por la prioridad de la tarea, que se establece en `addTask()`.

El estado de las tareas puede ser monitorizado usando señales y funciones `QgsTask` y `QgsTaskManager`.

15.2 Ejemplos

15.2.1 Extendiendo QgsTask

En este ejemplo `RandomIntegerSumTask` se extiende `QgsTask` y generará 100 enteros aleatorios entre 0 y 500 durante un período de tiempo específico. Si el número aleatorio es 42, la tarea se cancela y se genera una excepción. Se generan y agregan varias instancias de `RandomIntegerSumTask` (con subtareas) al administrador de tareas, lo que demuestra dos tipos de dependencias.

```
import random
from time import sleep

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

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""

    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None

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

    def finished(self, result):
        """
        This function is automatically called when the task has
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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


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

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

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

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

15.2.2 Tarea desde función

Cree una tarea desde una función (`doSomething` en este ejemplo). El primer parámetro de la función contendrá el `QgsTask` para la función. Un importante parámetro(llamado) es `on_finished`, que especifica una función que se llamará cuando la tarea se haya completado. La función `doSomething` en este ejemplo tiene un parámetro adicional llamado `wait_time`.

```
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```
        MESSAGE_CATEGORY, Qgis.Info)

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

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

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

15.2.3 Tarea de un algoritmo de procesamiento

Crear una tarea que use el algoritmo qgis:randompointsinextent para generar 50000 puntos aleatorios dentro de una extensión específica. El resultado se agrega al proyecto de manera segura.

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

MESSAGE_CATEGORY = 'AlgRunnerTask'

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

(continúe en la próxima página)

(proviene de la página anterior)

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

alg = QgsApplication.processingRegistry().algorithmById(
                                     'qgis:randompointsinextent')
# `context` and `feedback` need to
# live for as least as long as `task`,
# otherwise the program will crash.
# Initializing them globally is a sure way
# of avoiding this unfortunate situation.
context = QgsProcessingContext()
feedback = QgsProcessingFeedback()
params = {
    'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
    'MIN_DISTANCE': 0.0,
    'POINTS_NUMBER': 50000,
    'TARGET_CRS': 'EPSG:4326',
    'OUTPUT': 'memory:My random points'
}
task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
task.executed.connect(partial(task_finished, context))
QgsApplication.taskManager().addTask(task)
```

Vea también: https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/.

CAPÍTULO 16

Desarrollando Plugins Python

16.1 Estructurar Complementos de Python

Los pasos principales para crear un complemento son:

1. *Idea*: Tenga una idea de lo que quiere hacer con tu nuevo complemento QGIS.
2. *Configuración*: *Crea los archivos para tu complemento*. Dependiendo del tipo de complemento, algunos son obligatorios y otros opcionales.
3. *Desarrollar*: *Escribir el código* en los archivos correspondientes
4. *Documentar*: *Escribir la documentación del complemento*
5. Opcionalmente: * Traducir*: *Traduce su complemento* en diferentes idiomas
6. *Test*: *Recarga tu complemento* para comprobar si todo está bien
7. *Publish*: Publique su complemento en el repositorio de QGIS o cree su propio repositorio como un «arsenal» de «armas GIS» personales.

16.1.1 Cómo empezar

Antes de empezar a escribir un nuevo complemento, eche un vistazo al *Repositorio oficial de complementos de Python*. El código fuente de los complementos existentes puede ayudarle a aprender más sobre programación. También puedes encontrar que ya existe un complemento similar y puede ser capaz de extenderlo o al menos basarte en él para desarrollar el tuyo propio.

Configurar la estructura de archivos del complemento

Para empezar con un nuevo complemento, necesitamos configurar los archivos necesarios del complemento.

Existen dos recursos de plantillas de complementos que pueden ayudarle a empezar:

- Para fines educativos o siempre que se desee un enfoque minimalista, la plantilla minimal plugin template proporciona los archivos básicos (esqueleto) necesarios para crear un complemento válido de QGIS Python.
- Para una plantilla de complemento más completa, el Plugin Builder puede crear plantillas para múltiples tipos de complementos diferentes, incluyendo características como localización (traducción) y pruebas.

Un directorio de complementos típico incluye los siguientes archivos:

- `metadata.txt` - *requerido* - Contiene información general, versión, nombre y algunos otros metadatos utilizados por el sitio web de complementos y la infraestructura de complementos.
- `__init__.py` - *requerido* - El punto de inicio del complemento. Tiene que tener el método `classFactory()` y puede tener cualquier otro código de inicialización.
- `mainPlugin.py` - *código núcleo* - El código principal de trabajo del complemento. Contiene toda la información sobre las acciones del complemento y el código principal.
- `form.ui` - *para complementos con IGU personalizada* - La IGU creada por Qt Designer.
- `form.py` - *GUI compilado* - La traducción del form.ui descrito anteriormente a Python.
- `resources.qrc` - *opcional* - Un documento .xml creado por Qt Designer. Contiene rutas relativas a los recursos utilizados en los formularios GUI.
- `resources.py` - *recursos compilados, opcional* - La traducción del archivo .qrc descrito anteriormente a Python.

Advertencia: Si planea subir el complemento al *Repositorio oficial de complementos de Python* debe comprobar que su complemento sigue algunas reglas adicionales, requeridas para el complemento *Validación*.

16.1.2 Escribir el código del complemento

La siguiente sección muestra qué contenido debe añadirse en cada uno de los archivos introducidos anteriormente.

metadata.txt

En primer lugar, el Gestor de Complementos necesita recuperar cierta información básica sobre el complemento, como su nombre, descripción, etc. Esta información se almacena en `metadata.txt`.

Nota: Todos los metadatos deben estar en codificación UTF-8.

Nombre de metadatos	Neces	Notas
Nombre	Verdac	una cadena corta que contiene el nombre del complemento
qgisMinimu	Verdac	notación con puntos de la versión mínima de QGIS
qgisMaximu	Falso	notación con puntos de la versión máxima de QGIS
descripción	Verdac	texto breve que describe el complemento, no se permite HTML
acerca	Verdac	texto más largo que describe el complemento en detalles, no se permite HTML
versión	Verdac	cadena corta con la notación punteada de la versión
autor	Verdac	nombre del autor
email	Verdac	correo electrónico del autor, que solo se muestra en el sitio web para los usuarios que han iniciado sesión, pero que es visible en el Administrador de complementos después de instalar el complemento
registro de cambios	Falso	cadena, puede ser de varias líneas, no se permite HTML
experimental	Falso	bandera booleana, `True` o `False` - `True` si esta versión es experimental
obsoleto	Falso	bandera booleana, `True` o `False`, se aplica a todo el complemento y no solo a la versión cargada
etiquetas	Falso	lista separada por comas, los espacios son permitidos dentro de etiquetas individuales
homepage	Falso	una URL válida que apunta a la página de inicio de su complemento
repositorio	Verdac	una URL válida al repositorio de código fuente
rastreador	Falso	Una URL válida a los tickets y reportes de errores
icono	Falso	un nombre de archivo o una ruta relativa (relativa a la carpeta base del paquete comprimido del complemento) de una imagen compatible con la web (PNG, JPEG)
categoría	Falso	una de las siguientes opciones: `Raster`, `Vector`, `Base de datos`, `Malla` y `Web`.
plugin_depe	Falso	Lista separada por comas de otros complementos a instalar, utiliza los nombres de los complementos procedentes del campo de nombre de sus metadatos
servidor	Falso	bandera booleana, `True` o `False`, determina si el complemento tiene una interfaz de servidor
hasProcessin	Falso	bandera booleana, `True` o `False`, determina si el complemento proporciona algoritmos de procesamiento

De forma predeterminada, los complementos se colocan en el menú *Complementos* (veremos en la siguiente sección cómo añadir una entrada de menú para tu complemento) pero también pueden colocarse en los menús *Raster*, *Vector*, *Base de datos*, *Malla* y *Web*.

Existe una entrada de metadatos de «categoría» correspondiente para especificar eso, por lo que el complemento se puede clasificar en consecuencia. Esta entrada de metadatos se utiliza como sugerencia para los usuarios y les dice dónde (en qué menú) se puede encontrar el complemento. Los valores permitidos para «categoría» son: Vectorial, Ráster, Base de datos o Web. Por ejemplo, si su complemento estará disponible en el menú *Ráster*, agréguelo a `metadata.txt`

```
category=Raster
```

Nota: Si *qgisMaximumVersion* está vacío, se establecerá automáticamente en la versión principal más *.99* cuando se cargue en el *Repositorio oficial de complementos de Python*.

Un ejemplo para este metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=3.0
```

(continúe en la próxima página)

(proviene de la página anterior)

```
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
; end of mandatory metadata

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

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

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

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

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

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

; Since QGIS 3.8, a comma separated list of plugins to be installed
; (or upgraded) can be specified.
; The example below will try to install (or upgrade) "MyOtherPlugin" version 1.12
; and any version of "YetAnotherPlugin".
; Both "MyOtherPlugin" and "YetAnotherPlugin" names come from their own metadata's
; name field
plugin_dependencies=MyOtherPlugin==1.12,YetAnotherPlugin
```

__init__.py

Este archivo es requerido por el sistema de importación de Python. Además, QGIS requiere que este archivo contenga una función `classFactory()`, que se llama cuando el complemento se carga en QGIS. Recibe una referencia a la instancia de: class: *QgisInterface <qgis.gui.QgisInterface>* y debe devolver un objeto de la clase de su complemento desde `mainplugin.py` — en nuestro caso se llama `TestPlugin` (ver más abajo). Así es como debería verse `__init __. Py`

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

(continúe en la próxima página)

(proviene de la página anterior)

```
# any other initialisation needed
```

mainPlugin.py

Aquí es donde ocurre la magia y así es como se ve la magia: (por ejemplo `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 form signal of the canvas
    self.iface.mapCanvas().renderComplete.disconnect(self.renderTest)

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

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

Las únicas funciones de complemento que deben existir en el archivo fuente del complemento principal (por ejemplo `mainPlugin.py`) son:

- `__init__` que da acceso a la interfaz QGIS
- `initGui()` llamado cuando se carga el complemento
- `unload()` llamado cuando se descarga el complemento

En el ejemplo anterior, se usó `addPluginToMenu()`. Esto agregará la acción de menú correspondiente al menú *Complementos*. Existen métodos alternativos para agregar la acción a un menú diferente. Aquí hay una lista de esos métodos:

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

Todos tienen la misma sintaxis que el método `addPluginToMenu()`.

Se recomienda agregar su menú de complementos a uno de esos métodos predefinidos para mantener la coherencia en la forma en que se organizan las entradas de los complementos. Sin embargo, puede agregar su grupo de menú personalizado directamente a la barra de menú, como lo demuestra el siguiente ejemplo:

```
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()
```

No olvide establecer `QAction` y `QMenu` `objectName` con un nombre específico para su complemento para que pueda personalizarse.

Aunque las acciones de ayuda y acerca de también se pueden añadir a su menú personalizado, un lugar conveniente para hacerlas disponibles es en el menú principal de QGIS *Ayuda* ► *Complementos*. Esto se hace usando el método `pluginHelpMenu()`.

```
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):
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Cuando trabaje en un complemento real, es aconsejable escribir el complemento en otro directorio (de trabajo) y crear un archivo MAKE que generará UI + archivos de recursos e instalará el complemento en su instalación de QGIS.

16.1.3 Documentación de complementos

La documentación del complemento se puede escribir como archivos de ayuda HTML. El módulo `qgis.utils` proporciona una función, `showPluginHelp()` que abrirá el navegador de archivos de ayuda, de la misma manera que otras ayudas de QGIS.

La función `showPluginHelp()` busca archivos de ayuda en el mismo directorio que el módulo de llamada. Buscará, a su vez, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` `y :file:`index.html`, mostrando el que encuentre primero. Aquí `ll_cc` es la configuración regional de QGIS. Esto permite que se incluyan múltiples traducciones de la documentación con el complemento.

La función `showPluginHelp()` también puede tomar parámetros packageName, que identifica un complemento específico para el que se mostrará la ayuda, nombre de archivo, que puede reemplazar» índice «en los nombres de los archivos que se buscan, y sección, que es el nombre de una etiqueta de anclaje html en el documento en el que se colocará el navegador.

16.1.4 Traducción de complementos

Con unos pocos pasos, puede configurar el entorno para la localización del complemento de modo que, según la configuración regional de su computadora, el complemento se cargue en diferentes idiomas.

Requisitos de Software

La forma más fácil de crear y administrar todos los archivos de traducción es instalar Qt Linguist. En un entorno GNU/Linux basado en Debian, puede instalarlo escribiendo:

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

Archivos y directorio

Cuando cree el complemento, encontrará la carpeta `i18n` dentro del directorio principal del complemento.

Todos los archivos de traducción deben estar dentro de este directorio.

archivo .pro

Primero debe crear un archivo `.pro`, que es un archivo *proyecto* que puede ser administrado por **Qt Linguist**.

En este archivo `.pro` debe especificar todos los archivos y formularios que desea traducir. Este archivo se utiliza para configurar los archivos de localización y las variables. Un posible archivo de proyecto, que coincide con la estructura de nuestro *complemento de ejemplo*:

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

Su complemento puede seguir una estructura más compleja y puede estar distribuido en varios archivos. Si este es el caso, tenga en cuenta que `pylupdate5`, el programa que usamos para leer el archivo `.pro` y actualizar la cadena traducible, no expande los caracteres comodín, por lo que debe colocar cada archivo explícitamente en el archivo `.pro`. Su archivo de proyecto podría verse así:

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

Además, el archivo `your_plugin.py` es el archivo que *llama* a todos los menús y submenús de su complemento en la barra de herramientas de QGIS y desea traducirlos todos.

Finalmente, con la variable *TRANSLATIONS* puede especificar los idiomas de traducción que desee.

Advertencia: Asegúrese de nombrar el archivo `ts` como `your_plugin_ + language + .ts`, de lo contrario, la carga del idioma fallará. Utilice el atajo de 2 letras para el idioma (**it** para italiano, **de** a alemán, etc.)

archivo .ts

Una vez que haya creado el `.pro`, estará listo para generar los archivos `.ts` para el(los) idioma(s) de su complemento.

Abra una terminal, vaya al directorio `your_plugin/i18n` y escriba:

```
pylupdate5 your_plugin.pro
```

debería ver el archivo(s) `your_plugin_language.ts`.

Abra el archivo `.ts` con **Qt Linguist** y comience a traducir.

archivo .qm

Cuando termine de traducir su complemento (si algunas cadenas no se completan, se utilizará el idioma de origen para esas cadenas), debe crear el archivo `.qm` (el archivo `.ts` compilado que se utilizará por QGIS).

Simplemente abra un cd de terminal en el directorio `your_plugin/i18n` y escriba:

```
lrelease your_plugin.ts
```

ahora, en el directorio `i18n` verá el archivo(s) `your_plugin.qm`.

Traducir usando Makefile

Alternativamente, puede usar el archivo MAKE para extraer mensajes del código Python y los cuadros de diálogo Qt, si creó su complemento con Plugin Builder. Al comienzo del Makefile hay una variable LOCALES:

```
LOCALES = en
```

Agregue la abreviatura del idioma a esta variable, por ejemplo, para el idioma húngaro:

```
LOCALES = en hu
```

Ahora puede generar o actualizar el archivo `hu.ts` (y el `en.ts` también) desde las fuentes mediante:

```
make transup
```

Después de esto, ha actualizado el archivo `.ts` para todos los idiomas configurados en la variable LOCALES. Utilice **Qt Linguist** para traducir los mensajes del programa. Al finalizar la traducción, el transcompile puede crear los archivos `.qm`:

```
make transcompile
```

Tienes que distribuir archivos `.ts`" con tu complemento.

Carga el complemento

Para ver la traducción de su complemento, abra QGIS, cambie el idioma (*Configuración -> Opciones -> General*) y reinicie QGIS.

Debería ver su complemento en el idioma correcto.

> **Advertencia:** Si cambia algo en su complemento (nuevas IU, nuevo menú, etc.), debe **generar nuevamente** la versión de actualización de los archivos `.ts` y `.qm`, así que vuelva a ejecutar el comando de arriba.

16.1.5 Compartir su complemento

QGIS aloja cientos de complementos en el repositorio de complementos. ¡Considere compartir el suyo! Ampliará las posibilidades de QGIS y la gente podrá aprender de su código. Todos los complementos alojados se pueden encontrar e instalar desde QGIS con el Administrador de complementos.

La información y los requisitos están aquí: plugins.qgis.org.

16.1.6 Consejos y Trucos

Recargador de Complemento

Durante el desarrollo de su complemento, con frecuencia deberá volver a cargarlo en QGIS para realizar pruebas. Esto es muy fácil con el complemento **Recargador de complementos**. Puedes encontrarlo con el Administrador de Complementos.

Automatice el empaquetado, la publicación y la traducción con qgis-plugin-ci

qgis-plugin-ci proporciona una interfaz de línea de comandos para realizar el empaquetado y despliegue automatizado de los complementos de QGIS en su ordenador, o utilizando la integración continua como Flujos de trabajo de GitHub o Gitlab-CI así como Transifex para la traducción.

Permite liberar, traducir, publicar o generar un archivo XML de repositorio de complementos a través de CLI o en acciones CI.

Acceder a complementos

Puede acceder a todas las clases de complementos instalados desde QGIS utilizando Python, que puede ser útil para fines de depuración.

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

Registro de Mensajes

Los complementos tienen su propia pestaña dentro de log_message_panel.

Archivo de Recursos

Algunos complementos utilizan archivos de recursos, por ejemplo `resources.qrc` que definen recursos para la interfaz gráfica de usuario, como iconos:

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

Es bueno usar un prefijo que no colisione con otros complementos o partes de QGIS, de lo contrario, podría obtener recursos que no deseaba. Ahora solo necesita generar un archivo Python que contendrá los recursos. Se hace con el comando **pyrcc5**:

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

Nota: En entornos Windows, intentar ejecutar **pyrcc5** desde el símbolo del sistema o Powershell probablemente resultará en el error» Windows no puede acceder al dispositivo, ruta o archivo especificado […] «. La solución más fácil es probablemente utilizar OSGeo4W Shell, pero si se siente cómodo modificando la variable de entorno PATH o especificando la ruta al ejecutable de forma explícita, debería poder encontrarla en `<Your QGIS Install Directory>\bin\pyrcc5.exe`.

16.2 Fragmentos de código

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

```
from qgis.core import (
    QgsProject,
    QgsApplication,
    QgsMapLayer,
)

from qgis.gui import (
    QgsGui,
    QgsOptionsWidgetFactory,
    QgsOptionsPageWidget,
    QgsLayerTreeEmbeddedWidgetProvider,
    QgsLayerTreeEmbeddedWidgetRegistry,
)

from qgis.PyQt.QtCore import Qt
from qgis.PyQt.QtWidgets import (
    QMessageBox,
    QAction,
    QHBoxLayout,
    QComboBox,
)
from qgis.PyQt.QtGui import QIcon
```

Esta sección cuenta con fragmentos de código para facilitar el desarrollo de complementos.

16.2.1 Cómo llamar a un método por un atajo de teclado

En el complemento añadir a la `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)
```

Para añadir `unload()`

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

El método que se llama cuando se presiona CTRL+I

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

También es posible permitir a los usuarios personalizar los atajos de teclado para las acciones proporcionadas. Esto se hace añadiendo:

```
# in the initGui() function
QgsGui.shortcutsManager().registerAction(self.key_action)

# and in the unload() function
QgsGui.shortcutsManager().unregisterAction(self.key_action)
```

16.2.2 Cómo reutilizar los iconos de QGIS

Debido a que son bien conocidos y transmiten un mensaje claro a los usuarios, es posible que a veces desee reutilizar los iconos de QGIS en su plugin en lugar de dibujar y establecer uno nuevo. Utilice el método `getThemeIcon()`.

Por ejemplo, para reutilizar el icono mActionFileOpen.svg disponible en el repositorio de código de QGIS:

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

`iconPath()` es otro método para llamar a los iconos de QGIS. Encuentre ejemplos de llamadas a iconos temáticos en Imágenes incrustadas en QGIS - Hoja de trucos.

16.2.3 Interfaz para complemento en el cuadro de diálogo de opciones

Puede agregar una pestaña de opciones de complementos personalizados a *Configuración -> Opciones*. Esto es preferible a agregar una entrada de menú principal específica para las opciones de su complemento, ya que mantiene todas las configuraciones de la aplicación QGIS y las configuraciones del complemento en un solo lugar que es fácil de descubrir y navegar por los usuarios.

El siguiente fragmento solo agregará una nueva pestaña en blanco para la configuración del complemento, lista para que la complete con todas las opciones y configuraciones específicas de su complemento. Puede dividir las siguientes clases en diferentes archivos. En este ejemplo, estamos agregando dos clases al archivo principal `mainPlugin.py`.

```
class MyPluginOptionsFactory(QgsOptionsWidgetFactory):

    def __init__(self):
        super().__init__()

    def icon(self):
        return QIcon('icons/my_plugin_icon.svg')

    def createWidget(self, parent):
        return ConfigOptionsPage(parent)


class ConfigOptionsPage(QgsOptionsPageWidget):

    def __init__(self, parent):
        super().__init__(parent)
        layout = QHBoxLayout()
        layout.setContentsMargins(0, 0, 0, 0)
        self.setLayout(layout)
```

Finalmente estamos agregando las importaciones y modificando la función `__init__`:

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


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

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

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


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

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

Truco: Añadir pestañas personalizadas al diálogo de propiedades de capa

Puede aplicar una lógica similar para agregar la opción personalizada del complemento al cuadro de diálogo de propiedades de la capa usando las clases `QgsMapLayerConfigWidgetFactory` y `QgsMapLayerConfigWidget`.

16.2.4 Incrustar widgets personalizados para capas en el árbol de capas

Además de los elementos habituales de simbología de capas que se muestran al lado o debajo de la entrada de la capa en el panel *Capas*, puede añadir sus propios widgets, permitiendo un acceso rápido a algunas acciones que se utilizan a menudo con una capa (configuración de filtrado, selección, estilo, actualización de una capa con un widget de botón, creación de un deslizador de tiempo basado en la capa o simplemente mostrar información adicional de la capa en una etiqueta, o …). Estos llamados **Widgets incrustados en el árbol de capas** están disponibles a través de la pestaña *Legend* de las propiedades de cada capa.

El siguiente fragmento de código crea un desplegable en la leyenda que muestra los estilos de capa disponibles para la capa, permitiendo cambiar rápidamente entre los diferentes estilos de capa.

```
class LayerStyleComboBox(QComboBox):
    def __init__(self, layer):
        QComboBox.__init__(self)
        self.layer = layer
        for style_name in layer.styleManager().styles():
            self.addItem(style_name)

        idx = self.findText(layer.styleManager().currentStyle())
        if idx != -1:
          self.setCurrentIndex(idx)

        self.currentIndexChanged.connect(self.on_current_changed)

    def on_current_changed(self, index):
        self.layer.styleManager().setCurrentStyle(self.itemText(index))

class LayerStyleWidgetProvider(QgsLayerTreeEmbeddedWidgetProvider):
    def __init__(self):
        QgsLayerTreeEmbeddedWidgetProvider.__init__(self)

    def id(self):
        return "style"

    def name(self):
        return "Layer style chooser"

    def createWidget(self, layer, widgetIndex):
        return LayerStyleComboBox(layer)

    def supportsLayer(self, layer):
        return True   # any layer is fine

provider = LayerStyleWidgetProvider()
QgsGui.layerTreeEmbeddedWidgetRegistry().addProvider(provider)
```

A continuación, desde la pestaña de propiedades *Leyenda* de una capa determinada, arrastre el `Selector de estilo de capa` de *Widgets disponibles* a *Widgets utilizados* para habilitar el widget en el árbol de capas. Los widgets incrustados SIEMPRE se muestran en la parte superior de sus subelementos de nodo de capa asociados.

Si quiere utilizar los widgets desde, por ejemplo, un complemento, puede añadirlos así:

```
layer = iface.activeLayer()
counter = int(layer.customProperty("embeddedWidgets/count", 0))
layer.setCustomProperty("embeddedWidgets/count", counter+1)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
layer.setCustomProperty("embeddedWidgets/{}/id".format(counter), "style")
view = self.iface.layerTreeView()
view.layerTreeModel().refreshLayerLegend(view.currentLegendNode())
view.currentNode().setExpanded(True)
```

16.3 Configuración IDE para escribir y depurar complementos

Aunque cada programador tiene su editor de texto/IDE preferido, aquí hay algunas recomendaciones para configurar IDE populares para escribir y depurar complementos de QGIS Python.

16.3.1 Complementos útiles para escribir complementos de Python

Algunos complementos son convenientes al escribir complementos de Python. Desde *Complementos -> Administrar e instalar complementos…*, instala:

- *Plugin reloader*: Esto le permitirá recargar un complemento y realizar nuevos cambios sin reiniciar QGIS.
- *First Aid*: Esto agregará una consola de Python y un depurador local para inspeccionar las variables cuando se genere una excepción desde un complemento.

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

16.3.2 Una nota sobre la configuración de su IDE en Linux y Windows

On Linux, Todo lo que normalmente se debe hacer es agregar las ubicaciones de la biblioteca QGIS a la variable de entorno `PYTHONPATH` del usuario. En la mayoría de las distribuciones, esto se puede hacer editando `~/.bashrc` or `~/.bash-profile` con la siguiente línea (probado en OpenSUSE Tumbleweed):

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

Guarde el archivo e implemente la configuración del entorno mediante el siguiente comando de shell:

```
source ~/.bashrc
```

On Windows, debe asegurarse de tener la misma configuración de entorno y utilizar las mismas bibliotecas e intérprete que QGIS. La forma más rápida de hacerlo es modificar el archivo por lotes de inicio de QGIS.

Si utilizó el instalador de OSGeo4W, puede encontrarlo en la carpeta `bin` de su instalación de OSGeo4W. Busca algo como `C:\OSGeo4W\bin\qgis-unstable.bat`.

16.3.3 Depuración con Pyscripter IDE (Windows)

Para usar Pyscripter IDE, esto es lo que tienes que hacer:

1. Haz una copia de `qgis-unstable.bat` y renómbrela `pyscripter.bat`.
2. Ábrelo en un editor. Y elimine la última línea, la que inicia QGIS.
3. Agregue una línea que apunte a su ejecutable de Pyscripter y agregue el argumento de línea de comando que establece la versión de Python que se utilizará
4. También agregue el argumento que apunta a la carpeta donde Pyscripter puede encontrar la dll de Python utilizada por QGIS, puede encontrar esto en la carpeta bin de su instalación de 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. Ahora, cuando haga doble clic en este archivo por lotes, se iniciará Pyscripter, con la ruta correcta.

Más popular que Pyscripter, Eclipse es una opción común entre los desarrolladores. En la siguiente sección, explicaremos cómo configurarlo para desarrollar y probar complementos.

16.3.4 Depurar con Eclipse y PyDev

Instalación

Para utilizar Eclipse, asegúrese de haber instalado lo siguiente

- Eclipse
- Complemento Aptana Studio 3 or PyDev
- QGIS 2.x
- Es posible que también desee instalar ** Remote Debug **, un complemento de QGIS. Por el momento, todavía es experimental, habilite *Complementos experimentales* en *Complementos -> Administrar e instalar complementos...* ► *Opciones* de antemano.

Para preparar su entorno para usar Eclipse en Windows, también debe crear un archivo por lotes y usarlo para iniciar Eclipse:

1. Localice la carpeta donde `qgis_core.dll` reside. Normalmente este es `C:\OSGeo4W\apps\qgis\bin`, pero si compiló su propia aplicación QGIS, esto está en su carpeta de compilación en `output/bin/RelWithDebInfo`
2. Localice su ejecutable `eclipse.exe`.
3. Cree el siguiente script y utilícelo para iniciar eclipse al desarrollar complementos de 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 Eclipse

1. En Eclipse, cree un nuevo proyecto. Puede seleccionar * Proyecto general * y vincular sus fuentes reales más adelante, por lo que realmente no importa dónde coloque este proyecto.

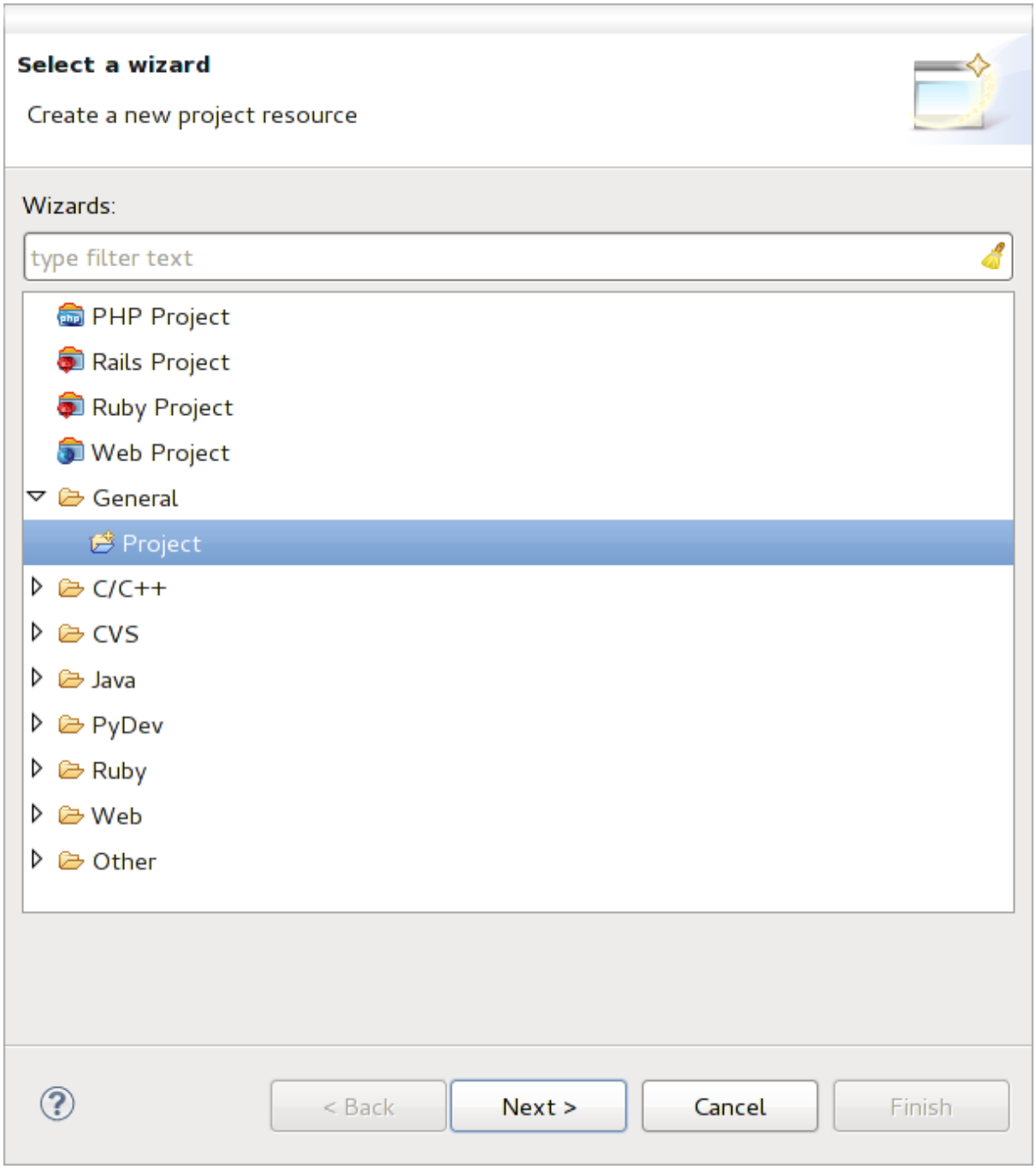


Figura 16.1: Proyecto Eclipse

2. Haz clic derecho en tu nuevo proyecto y elige *Nueva* ► *Carpeta*.
3. Click en *Acanzado* y elige *Enlace a ubicación alternativa (carpeta vinculada)*. En caso de que ya tenga fuentes que desee depurar, elija estas. En caso de que no lo haga, cree una carpeta como ya se explicó.

Ahora en la vista *Explorador de proyecto*, su árbol de fuentes aparece y puede comenzar a trabajar con el código. Ya tiene resaltado de sintaxis y todas las otras potentes herramientas IDE disponibles.

Configurar el depurador

Para que el depurador funcione:

1. Cambiar a la perspectiva de depuración en Eclipse (*Ventana* ► *Abrir Perspectiva* ► *Otra* ► *Depurar*).
2. inicie el servidor de depuración PyDev eligiendo *PyDev -> Iniciar servidor de depuración*.
3. Eclipse ahora está esperando una conexión de QGIS a su servidor de depuración y cuando QGIS se conecte al servidor de depuración, le permitirá controlar los scripts de Python. Eso es exactamente para lo que instalamos el complemento *Depuración remota*. Así que inicie QGIS en caso de que aún no lo haya hecho y haga clic en el símbolo de error.

Ahora puede establecer un punto de interrupción y tan pronto como el código lo golpee, la ejecución se detendrá y podrá inspeccionar el estado actual de su complemento. (El punto de interrupción es el punto verde en la imagen de abajo, establezca uno haciendo doble clic en el espacio en blanco a la izquierda de la línea donde desea que se establezca el punto de interrupción).

```
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 )
```

Figura 16.2: Breakpoint

Una cosa muy interesante que puede utilizar ahora es la consola de depuración. Asegúrese de que la ejecución esté detenida en un punto de interrupción antes de continuar.

1. Abra la vista de la consola (*Ventana -> Mostrar vista*). Mostrará la consola *Debug Server* que no es muy interesante. Pero hay un botón *Abrir consola* que le permite cambiar a una consola PyDev Debug más interesante.
2. Haga clic en la flecha junto al: guilabel: botón «Abrir consola» y seleccione * Consola PyDev *. Se abre una ventana para preguntarle qué consola desea iniciar.
3. Elija * PyDev Debug Console *. En caso de que esté atenuado y le indique que inicie el depurador y seleccione el marco válido, asegúrese de que tiene el depurador remoto adjunto y se encuentra actualmente en un punto de interrupción.

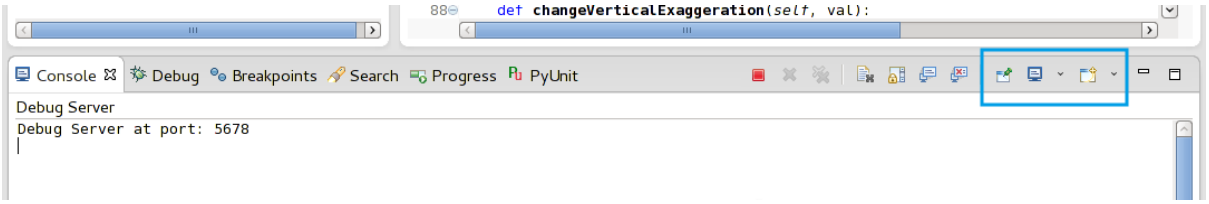


Figura 16.3: Consola de depuración de PyDev

Ahora tiene una consola interactiva que le permite probar cualquier comando desde el contexto actual. Puedes manipular variables o hacer llamadas a API o lo que quieras.

Truco: Un poco molesto es que cada vez que ingresa un comando, la consola vuelve al servidor de depuración. Para detener este comportamiento, puede hacer clic en el botón * Fijar consola * cuando esté en la página del servidor de depuración y debe recordar esta decisión al menos para la sesión de depuración actual.

Haciendo que eclipse entienda la API

Una característica muy útil es que Eclipse conozca realmente la API de QGIS. Esto le permite verificar su código en busca de errores tipográficos. Pero no solo esto, también permite que Eclipse lo ayude con el autocompletado desde las importaciones hasta las llamadas a la API.

Para hacer esto, Eclipse analiza los archivos de la biblioteca de QGIS y obtiene toda la información. Lo único que tienes que hacer es decirle a Eclipse dónde encontrar las bibliotecas.

1. Click *Ventana -> Preferencias -> PyDev -> Intérprete -> Python*.

   Verá su intérprete de Python configurado en la parte superior de la ventana (en este momento python2.7 para QGIS) y algunas pestañas en la parte inferior. Las pestañas más interesantes para nosotros son * Bibliotecas * y * Forced Builtins *.

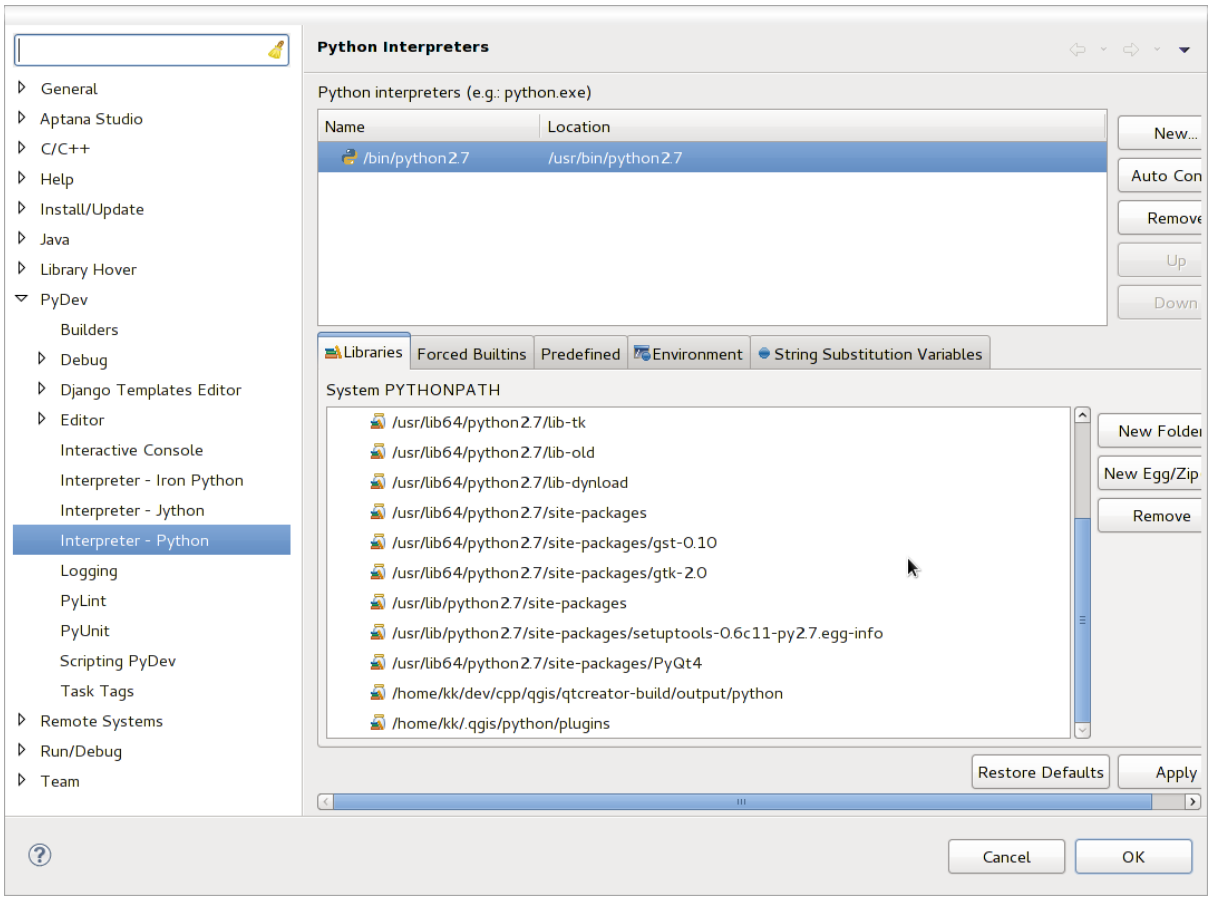


Figura 16.4: Consola de depuración de PyDev

2. Primero abra la pestaña Bibliotecas.
3. Agregue una nueva carpeta y elija la carpeta de Python de su instalación de QGIS. Si no sabe dónde está esta carpeta (no es la carpeta de complementos):
   1. Abrir QGIS
   2. Inicia una consola de python

3. Introduce `qgis`
4. y presione Entrar. Le mostrará qué módulo QGIS utiliza y su ruta.
5. Quite el `/qgis/__init__.pyc` final de esta ruta y obtendrá la ruta que está buscando.

4. También debe agregar su carpeta de complementos aquí (está en `python/complementos` debajo de la carpeta user profile).
5. Luego salte a la pestaña * Forced Builtins *, haga clic en * New … * e ingrese `` qgis “”. Esto hará que Eclipse analice la API de QGIS. Probablemente también desee que Eclipse conozca la API de PyQt. Por lo tanto, también agregue PyQt como incorporado forzado. Eso probablemente ya debería estar presente en la pestaña de bibliotecas.
6. Click en *Aceptar* y ya estas listo.

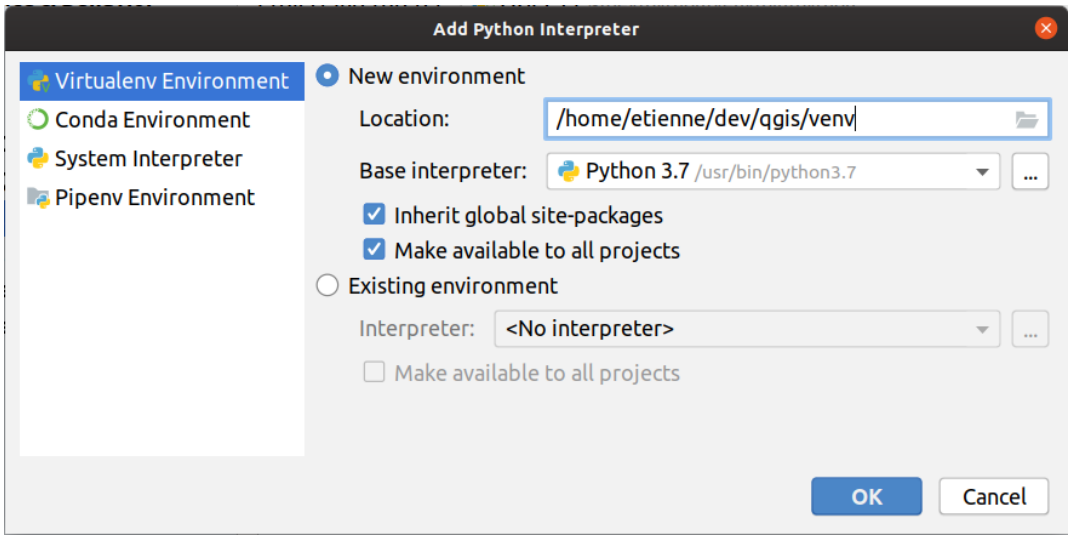
Nota: Cada vez que cambia la API de QGIS (por ejemplo, si está compilando el maestro QGIS y el archivo SIP cambió), debe volver a esta página y simplemente hacer clic en *Aplicar*. Esto permitirá que Eclipse vuelva a analizar todas las bibliotecas.

16.3.5 Depurar con PyCharm en Ubuntu con un QGIS compilado

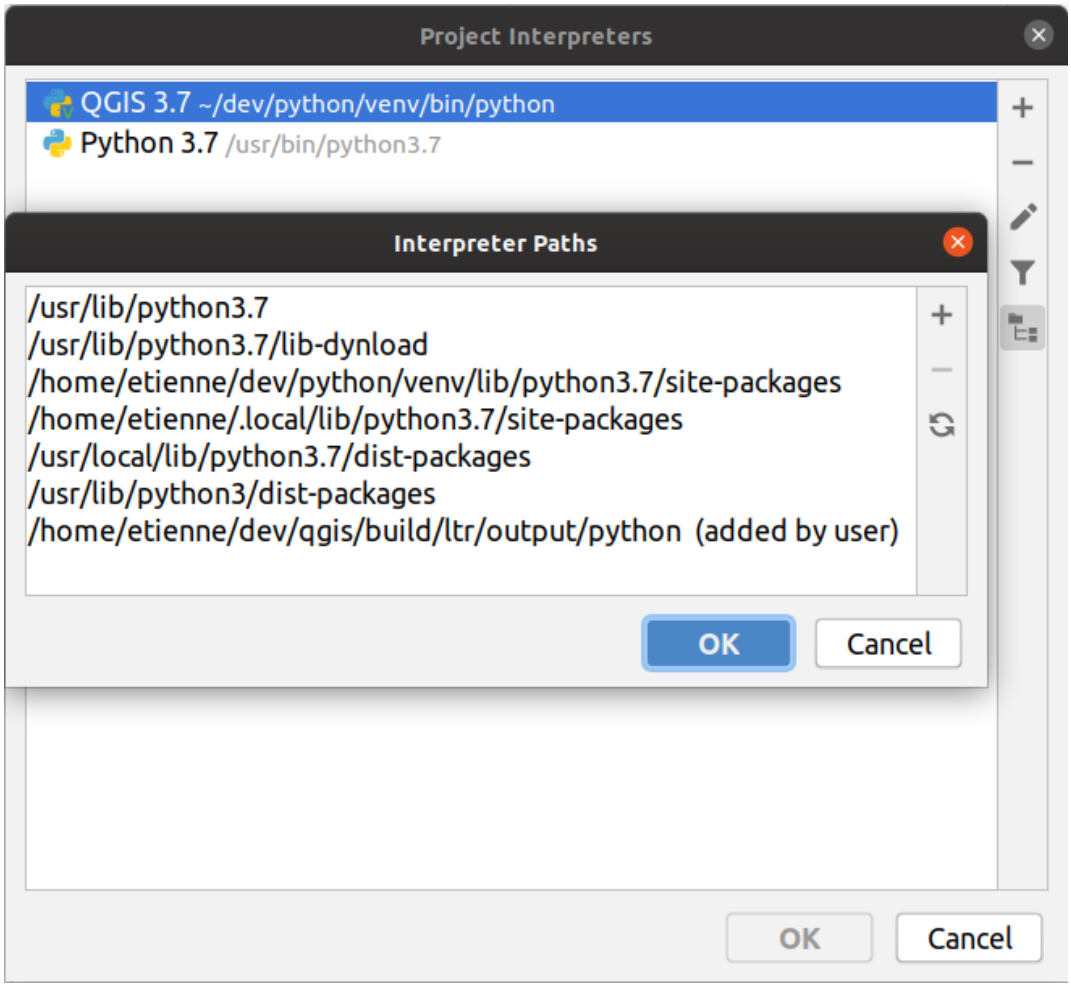
PyCharm es un IDE para Python desarrollado por JetBrains. Hay una versión gratuita llamada Community Edition y una de pago llamada Professional. Puede descargar PyCharm en el sitio web: https://www.jetbrains.com/pycharm/download

Suponemos que ha compilado QGIS en Ubuntu con el directorio de compilación proporcionado `~/dev/qgis/build/master`. No es obligatorio tener un QGIS autocompilado, pero solo esto ha sido probado. Los caminos deben adaptarse.

1. En PyCharm, en su: guilabel: *Propiedades del proyecto*, *Intérprete del proyecto*, vamos a crear un entorno virtual Python llamado `` QGIS``.
2. Haga clic en el engranaje pequeño y luego *Añadir*.
3. Selecciona *Virtualenv environment*.
4. Seleccione una ubicación genérica para todos sus proyectos de Python, como `~/dev/qgis/venv` porque usaremos este intérprete de Python para todos nuestros complementos.
5. Elija un intérprete base de Python 3 disponible en su sistema y marque las siguientes dos opciones *Inherit global site-packages* y *Poner a disposición de todos los proyectos*.



1. Click en *Aceptar*, regrese al pequeño engranaje y click en *Mostrar todo*.
2. En la nueva ventana, seleccione su nuevo intérprete `` QGIS “” y haga clic en el último icono del menú vertical *Mostrar rutas para el intérprete seleccionado*.
3. Finalmente, agregue la siguiente ruta absoluta a la lista `~/dev/qgis/build/master/output/python`.



1. Reinicie PyCharm y podrá comenzar a usar este nuevo entorno virtual de Python para todos sus complementos.

PyCharm conocerá la API de QGIS y también la API de PyQt si usa Qt proporcionado por QGIS como `from qgis.PyQt.QtCore import QDir`. El autocompletado debería funcionar y PyCharm puede inspeccionar su código.

En la versión profesional de PyCharm, la depuración remota funciona bien. Para la edición Community, la depuración remota no está disponible. Solo puede tener acceso a un depurador local, lo que significa que el código debe ejecutarse *dentro* de PyCharm (como script o unittest), no en QGIS. Para el código Python que se ejecuta *en* QGIS, puede usar el complemento *First Aid* mencionado anteriormente.

16.3.6 Depurar usando PDB

Si no usa un IDE como Eclipse o PyCharm, puede depurar usando PDB, siguiendo estos pasos.

1. Primero agregue este código en el lugar donde le gustaría 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. Luego ejecute QGIS desde la línea de comando.

   En Linux haga:

```
$ ./Qgis
```

   En macOS haga:

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

3. ¡Y cuando la aplicación llegue a su punto de interrupción, puede escribir en la consola!

PENDIENTE:
Agregar información de prueba

16.4 Lanzamiento de su complemento

Una vez que su complemento esté listo y crea que el complemento podría ser útil para algunas personas, no dude en cargarlo en *Repositorio oficial de complementos de Python*. En esa página también puede encontrar pautas de empaquetado sobre cómo preparar el complemento para que funcione bien con el instalador del complemento. O en caso de que desee configurar su propio repositorio de complementos, cree un archivo XML simple que enumere los complementos y sus metadatos.

Preste especial atención a las siguientes sugerencias:

16.4.1 Metadatos y nombres

- evite usar un nombre demasiado similar a los complementos existentes
- Si su complemento tiene una funcionalidad similar a un complemento existente, explique las diferencias en el campo Acerca de, para que el usuario sepa cuál usar sin la necesidad de instalarlo y probarlo.
- evitar repetir «complemento» en el nombre del propio complemento
- use el campo de descripción en los metadatos para una descripción de 1 línea, el campo Acerca de para obtener instrucciones más detalladas
- incluir un depósito de código, un rastreador de errores y una página de inicio; esto mejorará enormemente la posibilidad de colaboración y se puede hacer muy fácilmente con una de las infraestructuras web disponibles (GitHub, GitLab, Bitbucket, etc.)
- elija las etiquetas con cuidado: evite las que no son informativas (por ejemplo, vector) y prefiera las que ya utilizan otros (consulte el sitio web del complemento)
- agregue un icono adecuado, no deje el predeterminado; consulte la interfaz de QGIS para obtener una sugerencia del estilo que se utilizará

16.4.2 Código y ayuda

- no incluya el archivo generado (ui_*.py, resources_rc.py, archivos de ayuda generados …) y cosas inútiles (por ejemplo, .gitignore) en el repositorio
- agregue el complemento al menú apropiado (Vector, Raster, Web, Database)
- cuando sea apropiado (complementos que realizan análisis), considere agregar el complemento como un subplugin del marco de procesamiento: esto permitirá a los usuarios ejecutarlo en lotes, integrarlo en flujos de trabajo más complejos y lo liberará de la carga de diseñar una interfaz
- incluya al menos documentación mínima y, si es útil para probar y comprender, datos de muestra.

16.4.3 Repositorio oficial de complementos de Python

Puede encontrar el repositorio de complementos de Python *oficial* en https://plugins.qgis.org/.

Para utilizar el repositorio oficial, debe obtener un ID de OSGEO en el portal web OSGEO.

Una vez que haya cargado su complemento, un miembro del personal lo aprobará y se le notificará.

PENDIENTE:
Insertar un enlace al documento de gobernanza

Permisos

Estas reglas se han implementado en el repositorio oficial de complementos:

- cada usuario registrado puede agregar un nuevo complemento
- *staff* los usuarios pueden aprobar o rechazar todas las versiones del complemento
- los usuarios que tienen el permiso especial *plugins.can_approve* obtienen las versiones que cargan automáticamente aprobadas
- los usuarios que tienen el permiso especial *plugins.can_approve* pueden aprobar versiones cargadas por otros siempre que estén en la lista de plugins *propietarios*
- un complemento en particular puede ser eliminado y editado solo por *personal* usuarios y *propietarios de complementos*
- si un usuario sin el permiso *plugins.can_approve* carga una nueva versión, la versión del complemento no se aprueba automáticamente.

Gestión de confianza

Los miembros del personal pueden otorgar *confianza* a los creadores de complementos seleccionados que configuran el permiso *plugins.can_approve* a través de la aplicación frontal.

La vista de detalles del complemento ofrece enlaces directos para otorgar confianza al creador del complemento o los *propietarios* del complemento.

Validación

Los metadatos del complemento se importan y validan automáticamente desde el paquete comprimido cuando se carga el complemento.

Aquí hay algunas reglas de validación que debe conocer cuando desee cargar un complemento en el repositorio oficial:

1. el nombre de la carpeta principal que contiene su complemento debe contener solo caracteres ASCII (A-Z y a-z), dígitos y los caracteres de subrayado (_) y menos (-), además no puede comenzar con un dígito
2. `metadata.txt` es requerido
3. todos los metadatos requeridos enumerados en *tabla de metadatos* deben estar presentes
4. el campo de metadatos *versión* debe ser único

Estructura de complementos

Siguiendo las reglas de validación, el paquete comprimido (.zip) de su complemento debe tener una estructura específica para validarlo como un complemento funcional. Como el complemento se descomprimirá dentro de la carpeta de complementos de los usuarios, debe tener su propio directorio dentro del archivo .zip para no interferir con otros complementos. Los archivos obligatorios son: `metadata.txt` y `__init __.Py`. Pero sería bueno tener un: file: *README* y, por supuesto, un ícono para representar el complemento (`resources.qrc`). A continuación se muestra un ejemplo de cómo debería verse un plugin.zip.

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

Es posible crear complementos en el lenguaje de programación Python. En comparación con los complementos clásicos escritos en C++, estos deberían ser más fáciles de escribir, comprender, mantener y distribuir debido a la naturaleza dinámica del lenguaje Python.

Los complementos de Python se enumeran junto con los complementos de C++ en el administrador de complementos de QGIS. Se buscan en `~/(UserProfile)/python/plugins` and these paths:

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

Para definiciones de ~ y `(UserProfile)` vea core_and_external_plugins.

Nota: Al configurar *QGIS_PLUGINPATH* en una ruta de directorio existente, puede agregar esta ruta a la lista de rutas en las que se buscan complementos.

CAPÍTULO 17

Escribir nuevos complementos de procesamiento

Dependiendo del tipo de complemento que vaya a desarrollar, podría ser una mejor opción agregar su funcionalidad como un algoritmo de procesamiento (o un conjunto de ellos). Eso proporcionaría una mejor integración dentro de QGIS, funcionalidad adicional (ya que se puede ejecutar en los componentes de Processing, como el modelador o la interfaz de procesamiento por lotes), y un tiempo de desarrollo más rápido (ya que Processing tomará una gran parte del trabajo).

Para distribuir esos algoritmos, debe crear un nuevo complemento que los agregue a Caja de Herramientas de Procesos. El complemento debe contener un proveedor de algoritmos, que debe registrarse cuando se crea una instancia del complemento.

17.1 Creando desde cero

Para crear un complemento desde cero que contenga un proveedor de algoritmos, puede seguir estos pasos utilizando el Generador de complementos:

1. Instala el complemento **Plugin Builder**
2. Crea un nuevo complemento, usando el Plugin Builder. En el cuadro de diálogo del Plugin Builder, selecciona «Processing provider».
3. El complemento creado contiene un proveedor con un solo algoritmo. Tanto el archivo del proveedor como el archivo del algoritmo están completamente comentados y contienen información sobre cómo modificar el proveedor y agregar algoritmos adicionales. Consúltelos para obtener más información.

17.2 Actuializar un complemento

Si quiere añadir su complemento existente a Procesos, necesitará añadir algo de código.

1. En su archivo `metadata.txt`, necesitará añadir una variable:

   ```
   hasProcessingProvider=yes
   ```

2. En el archivo Python donde tu complemento está instalado con el método `initGui`, necesitas adaptar algunas líneas como esta:

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

class YourPluginName():

    def __init__(self):
        self.provider = None

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

    def initGui(self):
        self.initProcessing()

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

3. Puedes crear una carpeta `processing_provider` con tres archivos en ella:
   - `__init__.py` sin nada en él. Esto es necesario para crear un paquete Python válido.
   - `provider.py` que creará el proveedor de procesamiento y expondrá sus algoritmos.

```
from qgis.core import QgsProcessingProvider

from processing_provider.example_processing_algorithm import
↪ExampleProcessingAlgorithm


class Provider(QgsProcessingProvider):

    def loadAlgorithms(self, *args, **kwargs):
        self.addAlgorithm(ExampleProcessingAlgorithm())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())

    def id(self, *args, **kwargs):
        """The ID of your plugin, used for identifying the provider.

        This string should be a unique, short, character only string,
        eg "qgis" or "gdal". This string should not be localised.
        """
        return 'yourplugin'

    def name(self, *args, **kwargs):
        """The human friendly name of your plugin in Processing.

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

    def icon(self):
        """Should return a QIcon which is used for your provider inside
        the Processing toolbox.
        """
        return QgsProcessingProvider.icon(self)
```

   - `example_processing_algorithm.py` el cuál contiene el archivo de ejemplo de algoritmo. Copiar/pegar el contenido de la fuente archivo de plantilla de script y actualizalo de acuerdo a tus necesidades.

4. Ahora puede volver a cargar su complemento en QGIS y debería ver su script de ejemplo en la caja de herramientas de procesamiento y el modelador.

CAPÍTULO 18

Utilizar complemento Capas

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

```
from qgis.core import (
    QgsPluginLayer,
    QgsPluginLayerType,
    QgsMapLayerRenderer,
    QgsApplication,
    QgsProject,
)

from qgis.PyQt.QtGui import QImage
```

Si su complemento utiliza sus propios métodos para representar una capa de mapa, escribir su propio tipo de capa basándose en `QgsPluginLayer` podría ser la mejor manera de implementarlo.

18.1 Subclassing QgsPluginLayer

A continuación se muestra un ejemplo de una implementación mínima de QgsPluginLayer. Se basa en el código original del complemento de ejemplo Marca de agua.

El renderizador personalizado es la parte del implemento que define el dibujo real en el canvas.

```
class WatermarkLayerRenderer(QgsMapLayerRenderer):

    def __init__(self, layerId, rendererContext):
        super().__init__(layerId, rendererContext)

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

(continúe en la próxima página)

(proviene de la página anterior)

```
        return True

class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

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

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

    def setTransformContext(self, ct):
        pass

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

    def readXml(self, node, context):
        pass

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

La capa de complemento se puede agregar al proyecto y al lienzo como cualquier otra capa de mapa:

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

Al cargar un proyecto que contiene dicha capa, se necesita una clase de fábrica:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

    def __init__(self):
        super().__init__(WatermarkPluginLayer.LAYER_TYPE)

    def createLayer(self):
        return WatermarkPluginLayer()

    # You can also add GUI code for displaying custom information
    # in the layer properties
    def showLayerProperties(self, layer):
        pass


# Keep a reference to the instance in Python so it won't
# be garbage collected
plt =  WatermarkPluginLayerType()

assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)
```

CAPÍTULO 19

Biblioteca de análisis de redes

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

La biblioteca de análisis de red se puede utilizar para:

- crear un gráfico matemático a partir de datos geográficos (capas vectoriales de polilínea)
- implementar métodos básicos de la teoría de grafos (actualmente solo el algoritmo de Dijkstra)

La librería de análisis de redes fue creada por funciones básicas de exportación del complemento núcleo RoadGraph y ahora se puede utilizar los metodos en complementos o directamente de la consola Python.

19.1 Información general

Brevemente, un caso de uso típico se puede describir como:

1. Crear gráfica de geodatos (normalmente de capa vectorial de polilíneas)
2. ejecutar análisis gráfico
3. utilizar resultados de análisis (por ejemplo, visualizarlos)

19.2 Contruir un gráfico

Lo primero que hay que hacer — es preparar la entrada de datos, que es convertir una capa vectorial en un gráfico. Todas las acciones adicionales utilizarán esta gráfica, no la capa.

Como fuente podemos utilizar una capa vectorial de polilínea. Los nodos de las polilíneas se convierten en vértices del gráfico, y los segmentos de la polilínea son bordes de gráfico. Si varios nodos tienen la misma coordenada entonces ellos tienen el mimso vértice gráfico. Por lo que dos líneas que tienen un nodo en común se conectaran entre si.

Además durante la creación del gráfico se puede «arreglar» («atar») a la capa vectorial de entrada cualquier número de puntos adicionales. Para cada punto adicional se encontrará una coincidencia — el vértice gráfico más cercano o el borde gráfico más cercano. En el último caso el borde será dividido y un nuevo vértice se añadirá.

Los atributos de la capa vectorial y la longitud de un borde se puede utilizar como las propiedades de un borde.

La conversión de una capa vectorial al gráfico se realiza mediante el patrón de programación Constructor. Un gráfico se construye utilizando el llamado Director. Solo hay un Director por ahora: `QgsVectorLayerDirector`. El director establece la configuración básica que se utilizará para construir un gráfico a partir de una capa vectorial de línea, utilizada por el constructor para crear el gráfico. Actualmente, como en el caso del director, solo existe un constructor: `QgsGraphBuilder`, que crea objetos `QgsGraph`. Es posible que desee implementar sus propios constructores que crearán un gráfico compatible con bibliotecas como BGL o NetworkX.

Para calcular las propiedades del borde, se usó el patrón de programación estrategia. Por ahora solo estrategia `QgsNetworkDistanceStrategy` (que tiene en cuenta la longitud de la ruta) y `QgsNetworkSpeedStrategy` (que también considera la velocidad) están disponibles. Puede implementar su propia estrategia que utilizará todos los parámetros necesarios. Por ejemplo, el complemento RoadGraph utiliza una estrategia que calcula el tiempo de viaje utilizando la longitud del borde y el valor de velocidad de los atributos.

Es tiempo de sumergirse en el proceso.

En primer lugar, para utilizar esta biblioteca debemos importar el módulo de análisis

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

Después algunos ejemplos para crear un director

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

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

Para construir un director, debemos pasar una capa vectorial que se utilizará como fuente para la estructura del gráfico y la información sobre el movimiento permitido en cada segmento de la carretera (movimiento unidireccional o bidireccional, dirección directa o inversa). La llamada se ve así

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

Y aquí esta la lista completa de lo que significan estos parámetros:

- `vectorLayer` — capa vectorial usada para construir la gráfica

- `directionFieldId` — índice de la tabla de atributos de campo, donde se almacena información acerca de dirección de carreteras. Si `-1`, entonces no utilice esta información en absoluto. Un entero.
- `directDirectionValue` — el valor del campo de carreteras con dirección directa (mover desde la primer punto de línea a la última). Un texto.
- `reverseDirectionValue` — valor del campo de carreteras con dirección inversa (mover del último punto de línea al primero). Un texto.
- `bothDirectionValue` — valor de campo para carreteras bidireccionales (para cada carretera podemos mover del primer punto al último y del último al primero). Un texto.
- `defaultDirection` — dirección de la carretera predeterminada. Este valor se usará para aquellas carreteras donde el campo `directionFieldId` no está configurado o tiene algún valor diferente de cualquiera de los tres valores especificados anteriormente. Los posibles valores son:
  - `QgsVectorLayerDirector.DirectionForward` — dirección de un sentido
  - `QgsVectorLayerDirector.DirectionBackward` — Sentido único reversible
  - `QgsVectorLayerDirector.DirectionBoth` — Bi-direccional

Es necesario entonces crear una estrategia para calcular propiedades de borde

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

Y decirle al director sobre esta estrategia

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

Ahora podemos usar el constructor, que creará el gráfico. La clase `QgsGraphBuilder` constructor toma varios argumentos:

- `crs` — sistema de referencia de coordenadas a utilizar. Argumento obligatorio.
- `otfEnabled` — usar reproyección «al vuelo» o no. Por defecto `True` (usa OTF).
- `topologyTolerance` — tolerancia topológica. Por defecto es 0.
- `ellipsoidID` — elipsoide a usar. Por defecto «WGS84».

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

También podemos definir varios puntos, que se utilizarán en el análisis. Por ejemplo

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

Ahora todo está en su lugar para que podamos construir el gráfico y «atar» a estos puntos

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

Construir el grafo puede tomar tiempo (que depende del número de elementos y tamaño de una capa). `tiedPoints` es una lista con coordenadas de puntos «tied». Cuando la operación de construcción se finalizo podemos obtener la gráfica y utilizarlo para el análisis

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

Con el siguiente código podemos obtener el índice del vértice de nuestros puntos

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

19.3 Análisis gráfico

El análisis de redes es utilizado para encontrar respuestas a dos preguntas: que vértices estan conectados y cómo encontrar la ruta más corta. Para resolver estos problemas la librería de análisis de redes proporciona el algoritmo Dijkstra.

El algoritmo Dijkstra encuentra la ruta más corta de uno de los vértices del grafo a todos los otros y los valores de los parámetros de optimización, El resultado puede ser representado como un árbol de la ruta más corta.

El árbol de ruta más corto es un gráfico ponderado dirigido (o más precisamente un árbol) con las siguientes propiedades:

- sólo un vértice no tiene bordes entrantes — la raíz del árbol
- todos los otros vértices sólo tienen un borde entrante
- Si el vértice B es accesible desde el vértice A, entonces el camino de A a B es la única ruta disponible y es optima (más corta) en este grafo

Para obtener la ruta del árbol más corta, use los métodos `shortestTree()` y `dijkstra()` de la clase `QgsGraphAnalyzer` . Es recomendable usar el método `dijkstra()` porque funciona más rápido y usa la memoria de manera más eficiente.

El método `shortestTree()` es útil cuando desea caminar alrededor del árbol del camino más corto. Siempre crea un nuevo objeto gráfico (QgsGraph) y acepta tres variables:

- `source` — gráfica entrante
- `startVertexIdx` — índice del punto en el árbol (la raíz del árbol)
- `criterionNum` — número de propiedad de borde a usar (comenzando desde 0).

```
tree = QgsGraphAnalyzer.shortestTree(graph, startId, 0)
```

El método `dijkstra()` tiene los mismos argumentos, pero devuelve dos matrices. En el primer elemento de matriz, *n* contiene el índice del borde entrante o -1 si no hay bordes entrantes. En el segundo elemento de matriz, *n* contiene la distancia desde la raíz del árbol al vértice` n` o DOUBLE_MAX si el vértice *n* es inalcanzable desde la raíz.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, startId, 0)
```

Aquí hay un código muy simple para mostrar el árbol de ruta más corto usando el gráfico creado con el método `shortestTree()` (seleccione la capa de cadena de líneas en el panel *Capas* y reemplace las coordenadas por las suyas).

Advertencia: Use este código solo como ejemplo, crea muchos objetos `QgsRubberBand` y puede ser lento en conjuntos de datos extensos.

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
director = QgsVectorLayerDdirector(vectorLayer, -1, '', '', '',␣
```

(continúe en la próxima página)

(proviene de la página anterior)

```
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

i = 0
while (i < tree.edgeCount()):
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
  rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
  i = i + 1
```

Lo mismo pero usando el método `dijkstra()`

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')

director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
  rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())
```

19.3.1 Encontrar la ruta más corta

Para encontrar la ruta óptima entre dos puntos, se utiliza el siguiente enfoque. Ambos puntos (inicio A y final B) están «vinculados» al gráfico cuando se construye. Luego, usando el método `shortestTree ()` o `dijkstra()` construimos el árbol de ruta más corto con la raiz en el punto inicial A. En el mismo árbol también encontramos el punto final B y comenzamos a caminar a través del árbol desde el punto B al punto A. Todo el algoritmo se puede escribir como:

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

En este punto tenemos la ruta, en el formulario de la lista invertida de vértices (los vértices están listados en orden invertida del punto final al punto inicial) que serán visitados durante el viaje por este camino.

Aquí está el código de muestra para Consola de Pyhton QGIS (es posible que deba cargar y seleccionar una capa de cadena de líneas en TOC y reemplazar las coordenadas en el código por las suyas) que usa el método:meth:*shortestTree() <qgis.analysis.QgsGraphAnalyzer.shortestTree>*

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
builder = QgsGraphBuilder(vectorLayer.sourceCrs())
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)

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

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

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

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

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

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

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```
    route.insert(0, tree.vertex(edge.fromVertex()).point())
    idxEnd = edge.fromVertex()

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

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

Y aquí está la misma muestra pero usando el método `dijkstra()`

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *

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

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
↪'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)

builder = QgsGraphBuilder(vectorLayer.sourceCrs())

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

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

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

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

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

# Total cost
cost = costs[idxEnd]

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

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

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

# This may require coordinate transformation if project's CRS
```

(continúe en la próxima página)

(proviene de la página anterior)

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

19.3.2 Áreas de disponibilidad

El área de la disponibilidad para el vértice A es el subconjunto de vértices del grafo que son accesibles desde el vértice A y el costo de los caminos de la A a estos vértices son no es mayor que cierto valor.

Más claramente esto se puede demostrar con el siguiente ejemplo: «Hay una estación de bomberos ¿Qué partes de la ciudad puede un camión de bomberos alcanzar en 5 minutos? 10 minutos? 15 minutos?». Las respuestas a estas preguntas son las zonas de la estación de bomberos de la disponibilidad.

Para encontrar las áreas de disponibilidad podemos usar el método `dijkstra()` de la clase `QgsGraphAnalyzer`. Es suficiente comparar los elementos de la matriz de costos con un valor predefinido. Si el costo [i] es menor o igual a un valor predefinido, entonces el vértice i está dentro del área de disponibilidad, de lo contrario, está fuera.

Un problema más difícil es conseguir los límites de la zona de disponibilidad. El borde inferior es el conjunto de vértices que son todavía accesibles, y el borde superior es el conjunto de vértices que no son accesibles. De hecho esto es simple: es la frontera disponibilidad basado en los bordes del árbol de ruta más corta para los que el vértice origen del contorno es más accesible y el vértice destino del borde no lo es.

Aquí tiene un ejemplo

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',␣
↪QgsVectorLayerDirector.DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())


pStart = QgsPointXY(1179661.925139, 5419188.074362)
delta = iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.green)
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() + delta))

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

idStart = graph.findVertex(tStart)

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

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```
    i = i + 1

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

CAPÍTULO 20

Servidor QGIS y Python

20.1 Introducción

Para aprender mas sobre QGIS Server, lee QGIS-Server-manual.

QGIS Server es tres cosas diferentes:

1. Biblioteca de QGIS Server: una biblioteca que proporciona una API para crear servicios web OGC
2. QGIS Server FCGI: una aplicación binaria FCGI `qgis_mapserv.fcgi` que junto con un servidor web implementa un conjunto de servicios OGC (WMS, WFS, WCS etc.) y APIs OGC (WFS3/OAPIF)
3. QGIS Development Server: una aplicación binaria de servidor de desarrollo `qgis_mapserver` que implementa un conjunto de servicios OGC (WMS, WFS, WCS, etc.) y API de OGC (WFS3 / OAPIF)

Este capítulo del libro de cocina se centra en el primer tema y, al explicar el uso de la API de QGIS Server, muestra cómo es posible utilizar Python para ampliar, mejorar o personalizar el comportamiento del servidor o cómo utilizar la API de QGIS Server para incrustar el servidor QGIS en otra aplicación.

Hay algunas formas diferentes en las que puede alterar el comportamiento de QGIS Server o ampliar sus capacidades para ofrecer nuevos servicios personalizados o API, estos son los principales escenarios a los que puede enfrentarse:

- EMBEDDING → Utilice la API de QGIS Server desde otra aplicación de Python
- STANDALONE → Ejecutar QGIS Server como servicio autónomo WSGI/HTTP
- FILTERS → Mejorar/personalizar QGIS Server con complementos de filtro
- SERVICES → Añadir un nuevo *SERVICE*
- OGC APIs → Añadir una nueva *OGC API*

Las aplicaciones integradas e independientes requieren el uso de la API de Python de QGIS Server directamente desde otro script o aplicación de Python. Las opciones restantes son más adecuadas para cuando desee agregar funciones personalizadas a una aplicación binaria estándar de QGIS Server (FCGI o servidor de desarrollo): en este caso, deberá escribir un complemento de Python para la aplicación del servidor y registrar sus filtros personalizados, servicios o API.

20.2 Conceptos básicos de la API del servidor

Las clases fundamentales involucradas en una aplicación típica de QGIS Server son:

- `QgsServer` la instancia del servidor (normalmente una sola instancia durante toda la vida de la aplicación)
- `QgsServerRequest` el objeto de la solicitud (normalmente recreado en cada solicitud)
- `QgsServer.handleRequest(request, response)` procesa la solicitud y completa la respuesta

El flujo de trabajo del servidor de desarrollo o FCGI de QGIS Server se puede resumir de la siguiente manera:

```
initialize the QgsApplication
create the QgsServer
the main server loop waits forever for client requests:
    for each incoming request:
        create a QgsServerRequest request
        create a QgsServerResponse response
        call QgsServer.handleRequest(request, response)
            filter plugins may be executed
        send the output to the client
```

Dentro del métodod `QgsServer.handleRequest(request, response)` las devoluciones de llamada de los complementos de filtro se llaman y `QgsServerRequest` y `QgsServerResponse` están disponibles para los complementos a través de la clase `QgsServerInterface`

Advertencia: Las clases de servidor de QGIS no son seguras para subprocesos, siempre debe usar un modelo o contenedores de multiprocesamiento cuando cree aplicaciones escalables basadas en la API de QGIS Server.

20.3 Independiente o incrustado

Para aplicaciones de servidor independientes o incrustaciones, deberá usar las clases de servidor mencionadas anteriormente directamente, envolviéndolas en una implementación de servidor web que gestiona todas las interacciones del protocolo HTTP con el cliente.

A continuación, se muestra un ejemplo mínimo del uso de la API de QGIS Server (sin la parte HTTP):

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

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

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

# Create a response objects
response = QgsBufferServerResponse()

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```
23
24 app.exitQgis()
```

A continuación se muestra un ejemplo completo de aplicación independiente desarrollada para las pruebas de integraciones continuas en el repositorio de código fuente de QGIS, que muestra un amplio conjunto de diferentes filtros de complementos y esquemas de autenticación (no significa para la producción, ya que se desarrollaron sólo para fines de prueba, pero sigue siendo interesante para el aprendizaje): qgis_wrapped_server.py

20.4 Complementos del servidor

Los complementos de Python del servidor se cargan una vez cuando se inicia la aplicación QGIS Server y se pueden usar para registrar filtros, servicios o API.

La estructura de un complemento de servidor es muy similar a su contraparte de escritorio, un objeto `QgsServerInterface` está disponible para los complementos y los complementos pueden registrar uno o más filtros personalizados, servicios o API en el registro correspondiente mediante uno de los métodos expuestos por la interfaz del servidor.

20.4.1 Complementos de filtro de servidor

Los filtros vienen en tres sabores diferentes y se pueden instanciar subclasificando una de las clases a continuación y llamando al método correspondiente de `QgsServerInterface`:

Tipo de filtro	Clase Base	Registro de QgsServerInterface
E/S	`QgsServerFilter`	`registerFilter()`
Control de acceso	`QgsAccessControlFilter`	`registerAccessControl()`
Cache	`QgsServerCacheFilter`	`registerServerCache()`

Filtros de E/S

Los filtros de E/S pueden modificar la entrada y salida del servidor (la solicitud y la respuesta) de los servicios centrales (WMS, WFS, etc.) permitiendo realizar cualquier tipo de manipulación del flujo de trabajo de los servicios. Es posible, por ejemplo, restringir el acceso a las capas seleccionadas, inyectar una hoja de estilo XSL a la respuesta XML, agregar una marca de agua a una imagen WMS generada, etc.

A partir de este punto, encontrará útil un vistazo rápido a server plugins API docs.

Cada filtro debe implementar al menos una de las tres devoluciones de llamada:

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

Todos los filtros tienen acceso al objeto de solicitud/respuesta (`QgsRequestHandler`) y puede manipular todas sus propiedades (entrada/salida) y generar excepciones (aunque de una manera bastante particular como veremos a continuación).

Todos estos métodos devuelven un valor booleano que indica si la llamada debe propagarse a los filtros siguientes. Si uno de estos métodos devuelve `False` entonces la cadena se detiene, de lo contrario la llamada se propagará al siguiente filtro.

Aquí está el pseudocódigo que muestra cómo el servidor maneja una solicitud típica y cuándo se llaman las devoluciones de llamada del filtro:

```
for each incoming request:
    create GET/POST request handler
    pass request to an instance of QgsServerInterface
    call onRequestReady filters

    if there is not a response:
        if SERVICE is WMS/WFS/WCS:
            create WMS/WFS/WCS service
            call service's executeRequest
                possibly call onSendResponse for each chunk of bytes
                sent to the client by a streaming services (WFS)
        call onResponseComplete
    request handler sends the response to the client
```

Los siguientes párrafos describen las devoluciones de llamada disponibles en detalle.

onRequestReady

Esto se llama cuando la solicitud está lista: la URL entrante y los datos se han analizado y antes de ingresar al conmutador de servicios centrales (WMS, WFS, etc.), este es el punto donde puede manipular la entrada y realizar acciones como:

- autenticación/autorización
- redirije
- añadir/borrar ciertos parámetros (nombres de tipos, por ejemplo)
- plantear excepciones

Incluso podría sustituir un servicio central por completo cambiando el parámetro **SERVICE** y, por lo tanto, omitiendo el servicio central por completo (aunque esto no tiene mucho sentido).

onSendResponse

Esta llamada se realiza siempre que cualquier salida parcial se descarga desde el buffer de respuesta (es decir, a **FCGI** `stdout` si se utiliza el servidor fcgi) y desde allí, al cliente. Esto ocurre cuando se transmite un contenido enorme (como WFS GetFeature). En este caso `onSendResponse()` puede ser llamado varias veces.

Tenga en cuenta que si la respuesta no se transmite, entonces `onSendResponse()` no se llamará en absoluto.

En todos los casos, el último (o único) trozo se enviará al cliente tras una llamada a `onResponseComplete()`.

Devolver `False` prevendrá la descarga de datos al cliente. Esto es deseable cuando un complemento quiere recoger todos los trozos de una respuesta y examinar o cambiar la respuesta en `onResponseComplete()`.

onResponseComplete

Se llama una vez cuando los servicios centrales (si se han alcanzado) terminan su proceso y la petición está lista para ser enviada al cliente. Como se ha comentado anteriormente, este método se llamará antes de que el último (o único) trozo de datos se envíe al cliente. Para servicios de streaming, multiples llamadas a `onSendResponse()` pueden haber sido llamadas.

`onResponseComplete()` es el lugar ideal para proporcionar la implementación de nuevos servicios (WPS o servicios personalizados) y para realizar la manipulación directa de la salida procedente de los servicios centrales (por ejemplo para añadir una marca de agua sobre una imagen WMS).

Tenga en cuenta que devolver `False` evitará que los siguientes complementos ejecuten `onResponseComplete()` pero, en cualquier caso, evitará que se envíe la respuesta al cliente.

Generación de excepciones de un complemento

Aún queda trabajo por hacer en este tema: la implementación actual puede distinguir entre excepciones manejadas y no manejadas estableciendo una propiedad `QgsRequestHandler` a una instancia de QgsMapServiceException, de esta manera el código principal de C ++ puede detectar excepciones de Python controladas e ignorar las excepciones no controladas (o mejor: registrarlas).

Este enfoque básicamente funciona, pero no es muy «pitónico»: un mejor enfoque sería generar excepciones del código de Python y verlas burbujear en el bucle C ++ para que se manejen allí.

Escribiendo un complemento del servidor

Un complemento de servidor es un complemento estándar de QGIS Python como se describe en *Desarrollando Plugins Python*, que solo proporciona una interfaz adicional (o alternativa): un complemento de escritorio QGIS típico tiene acceso a la aplicación QGIS a través de: class:` QgisInterface <qgis. gui.QgisInterface> *, un complemento de servidor solo tiene acceso a :class:`QgsServerInterface <qgis.server.QgsServerInterface>* cuando se ejecuta dentro del contexto de la aplicación QGIS Server.

Para que QGIS Server sepa que un complemento tiene una interfaz de servidor, se necesita una entrada de metadatos especial (en `metadata.txt`):

```
server=True
```

Importante: QGIS Server solo cargará y ejecutará los complementos que tengan el conjunto de metadatos `server=True`.

El complemento de ejemplo qgis3-server-vagrant discutido aquí (con muchos más) está disponible en github, algunos complementos de servidor también están publicados en el repositorio oficial de complementos QGIS.

Archivos de complementos

Aquí está la estructura de directorios de nuestro complemento de servidor de ejemplo.

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

__init__.py

Este archivo es requerido por el sistema de importación de Python. Además, QGIS Server requiere que este archivo contenga una función `serverClassFactory()`, que se llama cuando el complemento se carga en QGIS Server cuando se inicia el servidor. Recibe una referencia a la instancia de `QgsServerInterface` y debe devolver la instancia de la clase de su complemento. Así es como se ve el complemento de ejemplo `__init __. Py`:

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

HelloServer.py

Aquí es donde ocurre la magia y así es como se ve la magia: (por ejemplo `HelloServer.py`)

Un complemento de servidor generalmente consiste en una o más devoluciones de llamada empaquetadas en instancias de `QgsServerFilter`.

Cada `QgsServerFilter` implementa una o más de las siguientes devoluciones de llamada:

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

El siguiente ejemplo implementa un filtro mínimo que imprime *HelloServer!* En caso de que el parámetro **SERVICE** sea igual a «HELLO»:

```
class HelloFilter(QgsServerFilter):

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

    def onRequestReady(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onRequestReady")
        return True

    def onSendResponse(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onSendResponse")
        return True

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

Los filtros deben registrarse en **serverIface** como en el siguiente ejemplo:

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

El segundo parámetro del método `registerFilter()` establece una prioridad que define el orden de las devoluciones de llamada con el mismo nombre (la prioridad más baja se invoca primero).

Al utilizar las tres devoluciones de llamada, los complementos pueden manipular la entrada y / o la salida del servidor de muchas formas diferentes. En todo momento, la instancia del complemento tiene acceso a `QgsRequestHandler` a través de `QgsServerInterface`. La clase `QgsRequestHandler` tiene muchos métodos que se pueden usar para modificar los parámetros de entrada antes de ingresar al procesamiento central del servidor (usando `requestReady()`) o después de la solicitud ha sido procesado por los servicios centrales (usando `sendResponse()`).

Los siguientes ejemplos cubren algunos casos comunes de uso:

Modificando la entrada

El complemento de ejemplo contiene un ejemplo de prueba que cambia los parámetros de entrada provenientes de la cadena de consulta, en este ejemplo se inyecta un nuevo parámetro en el (ya analizado) `parameterMap`, este parámetro es visible luego por los servicios centrales (WMS, etc.) , al final del procesamiento de los servicios centrales, verificamos que el parámetro todavía esté allí:

```
class ParamsFilter(QgsServerFilter):

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

    def onRequestReady(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')
        return True

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.onResponseComplete")
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.onResponseComplete")
        return True
```

Esto es un extracto de lo que puede ver en el archivo de log:

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

En la línea resaltada, la cadena «SUCCESS» indica que el complemento pasó la prueba.

La misma técnica se puede aprovechar para usar un servicio personalizado en lugar de uno central: por ejemplo, podría omitir una solicitud **SERVICE WFS** o cualquier otra solicitud principal simplemente cambiando el parámetro **SERVICE** a algo diferente y se omitirá el servicio principal. Luego, puede inyectar sus resultados personalizados en la salida y enviarlos al cliente (esto se explica a continuación).

Truco: Si realmente desea implementar un servicio personalizado, se recomienda crear una subclase `QgsService` y registre su servicio en `registerFilter()` llamando a su método `registerService(service)`

Modificar o reemplazar la salida

El ejemplo del filtro de marca de agua muestra cómo reemplazar la salida de WMS con una nueva imagen obtenida agregando una imagen de marca de agua en la parte superior de la imagen de WMS generada por el servicio principal de WMS:

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

class WatermarkFilter(QgsServerFilter):

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

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (params.get('SERVICE').upper() == 'WMS' \
                and params.get('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.onResponseComplete: image ready %s" % request.parameter("FORMAT"))
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG" if "png" in request.parameter("FORMAT") else "JPG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
        return True
```

En este ejemplo, el valor del parámetro **SERVICE** se verifica y si la solicitud entrante es un **WMS GETMAP** y no se han establecido excepciones por un complemento ejecutado previamente o por el servicio central (WMS en este caso), la imagen generada por WMS se recupera del búfer de salida y se agrega la imagen de marca de agua. El último paso es borrar el búfer de salida y reemplazarlo con la imagen recién generada. Tenga en cuenta que, en una situación del mundo real, también deberíamos comprobar el tipo de imagen solicitada en lugar de admitir solo PNG o JPG.

Filtros de control de acceso

Los filtros de control de acceso brindan al desarrollador un control detallado sobre las capas, características y atributos a los que se puede acceder; las siguientes devoluciones de llamada se pueden implementar en un filtro de control de acceso:

- `layerFilterExpression(layer)`
- `layerFilterSubsetString(layer)`
- `layerPermissions(layer)`
- `authorizedLayerAttributes(layer, attributes)`

- `allowToEdit(layer, feature)`
- `cacheKey()`

Archivos de complementos

Aquí está la estructura de directorios de nuestro complemento de ejemplo:

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

__init__.py

Este archivo es requerido por el sistema de importación de Python. Como para todos los complementos del servidor QGIS, este archivo contiene una función `serverClassFactory ()`, que se llama cuando el complemento se carga en el servidor QGIS al inicio. Recibe una referencia a una instancia de `QgsServerInterface` y debe devolver una instancia de la clase de su complemento. Así es como el complemento de ejemplo `__init__.py` aparece:

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

AccessControl.py

```
class AccessControlFilter(QgsAccessControlFilter):

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

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

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

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

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

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

    def cacheKey(self):
        return super().cacheKey()

class AccessControlServer:
```

(continúe en la próxima página)

(proviene de la página anterior)

```
30
31     def __init__(self, serverIface):
32         """ Register AccessControlFilter """
33         serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

Este ejemplo otorga acceso total para todos.

Es rol del complemento saber quién ha ingresado.

En todos esos métodos tenemos el argumento layer on para poder personalizar la restricción por capa.

layerFilterExpression

Permite añadir una expresión para limitar los resultados.

Por ejemplo, para limitar a características donde el atributo `role` es igual a `user`.

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

layerFilterSubsetString

Igual que el anterior pero usa el `SubsetString` (ejecutado en la base de datos)

Por ejemplo, para limitar a características donde el atributo `role` es igual a `user`.

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

layerPermissions

Limitar el acceso a la capa.

Devuelve un objeto de tipo `LayerPermissions()`, el cuál tiene las propiedades:

- `canRead` para verlo en `GetCapabilities` y tiene acceso de lectura.
- `canInsert` para poder insertar una nueva característica.
- `canUpdate` para ser capaz de actualizar una característica.
- `canDelete` para ser capaz de borrar una característica.

Por ejemplo, para limitar todo al acceso de sólo lectura:

```
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

Usado para limitar la visibilidad de un subconjunto específico de atributo.

El atributo del argumento devuelve el conjunto actual de atributos visibles.

Por ejemplo, para ocultar el atributo `role`:

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

allowToEdit

Esto es usado para limitar la edición de un subconjunto de objetos espaciales.

Se utiliza en el protocolo `WFS-Transaction`.

Por ejemplo, para poder editar sólo el objeto espacial que tiene el atributo `role` con el valor `user`:

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

cacheKey

QGIS Server mantiene una caché de las capacidades entonces para tener una caché por rol puedes retornar el rol en este método. O devolver `None` para desactivar completamente la caché.

20.4.2 Servicios personalizados

En QGIS Server, los servicios centrales como WMS, WFS y WCS se implementan como subclases de `QgsService`.

Para implementar un nuevo servicio que se ejecutará cuando el parámetro de cadena de consulta `SERVICE` coincida con el nombre del servicio, puedes implementar tu propio `QgsService` y registrar tu servicio en el `serviceRegistry()` llamando a su `registerService(service)`.

Este es un ejemplo de un servicio personalizado llamado `CUSTOM`:

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

class CustomServiceService(QgsService):

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

    def name(self):
        return "CUSTOM"

    def version(self):
        return "1.0.0"

    def executeRequest(self, request, response, project):
        response.setStatusCode(200)
        QgsMessageLog.logMessage('Custom service executeRequest')
        response.write("Custom service executeRequest")


class CustomService():

```

(continúe en la próxima página)

(proviene de la página anterior)

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

20.4.3 APIs personalizadas

En QGIS Server, las API principales de OGC como OAPIF (también conocido como WFS3) se implementan como colecciones de `QgsServerOgcApiHandler` subclases que están registradas en una instancia de `QgsServerOgcApi`).

Para implementar una nueva API que se ejecute cuando la ruta url coincida con una determinada URL, puede implementar sus propias instancias `QgsServerOgcApiHandler`, añadirlas a una `QgsServerOgcApi` y registrar la API en el `serviceRegistry()` llamando a su `registerApi(api)`.

A continuación, se muestra un ejemplo de una API personalizada que se ejecutará cuando la URL contenga `/customapi`:

```
import json
import os

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

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


class CustomApiHandler(QgsServerOgcApiHandler):

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

    def path(self):
        return QRegularExpression("/customapi")

    def operationId(self):
        return "CustomApiXYCircle"

    def summary(self):
        return "Creates a circle around a point"

    def description(self):
        return "Creates a circle around a point"

    def linkTitle(self):
        return "Custom Api XY Circle"

```

(continúe en la próxima página)

(proviene de la página anterior)

```
    def linkType(self):
        return QgsServerOgcApi.data

    def handleRequest(self, context):
        """Simple Circle"""

        values = self.values(context)
        x = values['x']
        y = values['y']
        r = values['r']
        f = QgsFeature()
        f.setAttributes([x, y, r])
        f.setGeometry(QgsCircle(QgsPoint(x, y), r).toCircularString())
        exporter = QgsJsonExporter()
        self.write(json.loads(exporter.exportFeature(f)), context)

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

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


class CustomApi():

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

CAPÍTULO 21

hoja de referencia para PyQGIS

Consejo: Los fragmentos de código en esta página necesitan las siguientes adiciones si está fuera de la consola de pyqgis:

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

from qgis.core import (
    Qgis,
    QgsProject,
    QgsLayerTreeModel,
)

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

21.1 Interfaz de Usuario

Cambiar apariencia

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
app.setStyleSheet(".QWidget {color: blue; background-color: yellow;}")
# You can even read the stylesheet from a file
with  open("testdata/file.qss") as qss_file_content:
    app.setStyleSheet(qss_file_content.read())
```

Cambiar icono y título

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon("/path/to/logo/file.png")
```

(continúe en la próxima página)

(proviene de la página anterior)

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

21.2 Configuración

Obtenga la lista de QgsSettings

```
from qgis.core import QgsSettings

qs = QgsSettings()

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

21.3 Barras de herramientas

Eliminar barra de herramientas

```
toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)

# and add again
parent.addToolBar(toolbar)
```

Eliminar acciones de un barra de herramientas

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

21.4 Menús

Eliminar menú

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

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

21.5 Lienzo

Acceder al lienzo

```
canvas = iface.mapCanvas()
```

Cambiar color de lienzo

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

Intervalo de actualización del mapa

```
from qgis.core import QgsSettings
# Set milliseconds (150 milliseconds)
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

21.6 Capas

Añadir capa vectorial

```
layer = iface.addVectorLayer("testdata/airports.shp", "layer name you like", "ogr")
if not layer or not layer.isValid():
    print("Layer failed to load!")
```

Obtener capa activa

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

Listar todas las capas

```
from qgis.core import QgsProject

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

Obtener el nombre de las capas

```
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']
```

De otra manera

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

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

Encuentra una capa por el nombre

```
from qgis.core import QgsProject

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

```
layer name you like
```

Establecer capa activa

```
from qgis.core import QgsProject

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

Intervalo para actualizar capa

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)
# Enable data reloading
layer.setAutoRefreshMode(Qgis.AutoRefreshMode.ReloadData)
```

Mostrar métodos

```
dir(layer)
```

Agregar nueva objeto espacial con un formulario

```
from qgis.core import QgsFeature, QgsGeometry

feat = QgsFeature()
geom = QgsGeometry()
feat.setGeometry(geom)
feat.setFields(layer.fields())

iface.openFeatureForm(layer, feat, False)
```

Agregar nueva objeto espacial sin un formulario

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

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

Obtener los objetos espaciales

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

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

Obtener los objetos espaciales seleccionados

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

Obtener los Ids de los objetos espaciales seleccionados

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

Crear una capa en memoria a partir de los Ids seleccionados

```
from qgis.core import QgsFeatureRequest

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

Obtener geometría

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

```
10.000000, 10.000000
```

Mover geometría

```
from qgis.core import QgsFeature, QgsGeometry
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POINT(7 45)")
geom.translate(1, 1)
poly.setGeometry(geom)
print(poly.geometry())
```

```
<QgsGeometry: Point (8 46)>
```

Establecer SRC

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

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

Ver SRC

```
from qgis.core import QgsProject

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

Ocultar columna

```
from qgis.core import QgsEditorWidgetSetup

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

(continúe en la próxima página)

(proviene de la página anterior)

```
        else:
            continue
```

Capa desde WKT

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

layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.09 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))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])
```

** Cargue todas las capas vectoriales de GeoPackage **

```
from qgis.core import QgsDataProvider

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

for subLayer in subLayers:
    name = subLayer.split(QgsDataProvider.SUBLAYER_SEPARATOR)[1]
    uri = "%s|layername=%s" % (fileName, name,)
    # Create layer
    sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
    # Add layer to map
    QgsProject.instance().addMapLayer(sub_vlayer)
```

Cargar capa de mosaico (XYZ-Layer)

```
from qgis.core import QgsRasterLayer, QgsProject

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

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

Eliminar todas las capas

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

Eliminar todos

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

21.7 Tabla de contenidos

Accede a las capas activadas

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

Eliminar menú contextual

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

21.8 TOC avanzado

Nodo raíz

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

root = QgsProject.instance().layerTreeRoot()
node_group = root.addGroup("My Group")

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

node_group.addLayer(layer)

print(root)
print(root.children())
```

Acceder al primer nodo hijo

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

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

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

Encontrar grupos y nodos

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer

def get_group_layers(group):
   print('- group: ' + group.name())
   for child in group.children():
      if isinstance(child, QgsLayerTreeGroup):
         # Recursive call to get nested groups
         get_group_layers(child)
      else:
         print('  - layer: ' + child.name())
```

(continúe en la próxima página)

(proviene de la página anterior)

```

root = QgsProject.instance().layerTreeRoot()
for child in root.children():
   if isinstance(child, QgsLayerTreeGroup):
      get_group_layers(child)
   elif isinstance(child, QgsLayerTreeLayer):
      print ('- layer: ' + child.name())
```

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

Buscar grupo por nombre

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

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

Encontrar capa por id

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

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

Añadir capas

```
from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
# Remove it
QgsProject.instance().removeMapLayer(layer1)
```

Añadir grupo

```
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
QgsProject.instance().mapLayersByName("layer name you like")[0]
```

Mover capa cargada

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

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

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

parent.removeChildNode(myLayer)
```

Mover capa cargada a un grupo específico

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

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

Alternar la visibilidad de la capa activa

```
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)
```

Es el grupo seleccionado

```
def isMyGroupSelected( groupName ):
    myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
    return myGroup in iface.layerTreeView().selectedNodes()

print(isMyGroupSelected( 'my group name' ))
```

```
False
```

Expandir nodo

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

Truco para ocultar nodo

```
from qgis.core import QgsProject

model = iface.layerTreeView().layerTreeModel()
ltv = iface.layerTreeView()
root = QgsProject.instance().layerTreeRoot()

layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
node = root.findLayer(layer.id())

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

Señales de nodo

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

def onAddedChildren(node, indexFrom, indexTo):
    print ("ADDED", node, indexFrom, indexTo)

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

Eliminar capa

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

Eliminar grupo

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

Crear nueva tabla de contenido (TOC)

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

Mover nodo

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

Cambiar nombre del nodo

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

21.9 Algoritmos de procesamiento

Obtener listado de algoritmos

```
from qgis.core import QgsApplication

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

```
QGIS (native c++):buffer --> Buffer
```

Obtener ayuda de los algoritmos

Selección aleatoria

```
from qgis import processing
processing.algorithmHelp("native:buffer")
```

```
...
```

Ejecutar algoritmo

Para este ejemplo, el resultado se almacena en una capa en memoria que se agrega al proyecto.

```
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'}
```

¿Cuántos algoritmos hay?

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

¿Cuántos proveedores hay?

```
from qgis.core import QgsApplication

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

¿Cuántas expresiones hay?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

21.10 Decoradores

CopyRight

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

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

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

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

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

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

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

```

(continúe en la próxima página)

(proviene de la página anterior)

```
    elif case == 1:
        # Bottom Left
        yOffset = deviceHeight - yOffset - size.height()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 2:
        # Top Right
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 3:
        # Bottom Right
        yOffset  = deviceHeight - yOffset - size.height()
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 4:
        # Top Center
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

    else:
        # Bottom Center
        yOffset = deviceHeight - yOffset - size.height()
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

# Emitted when the canvas has rendered
iface.mapCanvas().renderComplete.connect(_on_render_complete)
# Repaint the canvas map
iface.mapCanvas().refresh()
```

21.11 Compositor

Obtener diseño de impresión por nombre

```
composerTitle = 'MyComposer' # Name of the composer

project = QgsProject.instance()
projectLayoutManager = project.layoutManager()
layout = projectLayoutManager.layoutByName(composerTitle)
```

21.12 Fuentes

- QGIS Python (PyQGIS) API
- QGIS C++ API
- **`Preguntas de StackOverFlow sobre QGIS<https://stackoverflow.com/questions/tagged/qgis>`_**
- Script por Klas Karlsson