# PyQGIS developer cookbook

リリース *2.8*

**QGIS Project**

2016 年 07 月 30 日

# Contents

# Chapter 1

# はじめに

このドキュメントはチュートリアルとリファレンスガイドの両方の役割を意図して書かれています。すべてのユースケースを満たしてはいませんが、主要な機能の良い概要となるでしょう。

0.9 リリースから QGIS は Python を使ったスクリプト処理をサポートしました。Python はスクリプト処理でもっとも好まれている言語の一つだと思います。PyQGIS バインディングは SIP と PyQt4 に依存しています。これは SIP は SWIG の代わりに広く使われていて、QGIS のコードは Qt ライブラリに依存しています。Qt の Python バインディング (PyQt) も SIP を使っていて、これにより PyQt による PyQGIS の実装がシームレスに実現しています。

**TODO:** Getting PyQGIS to work (Manual compilation, Troubleshooting)

There are several ways how to use QGIS python bindings, they are covered in detail in the following sections:

- automatically run Python code when QGIS starts
- QGIS の中の Python コンソールのコマンドについて
- Python でプラグインを作り、使う方法
- QGIS API ベースのカスタムアプリケーションの作成

QGIS ライブラリのクラスのドキュメントは ‘完全な QGIS API <http://doc.qgis.org/>‘_ のリファレンスにあります。Python の QGIS API は C++ の API とほぼ同じです。

There are some resources about programming with PyQGIS on QGIS blog. See QGIS tutorial ported to Python for some examples of simple 3rd party apps. A good resource when dealing with plugins is to download some plugins from plugin repository and examine their code. Also, the `python/plugins/` folder in your QGIS installation contains some plugin that you can use to learn how to develop such plugin and how to perform some of the most common tasks

## 1.1 Run Python code when QGIS starts

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

### 1.1.1 PYQGIS_STARTUP environment variable

You can run Python code just before QGIS initialization completes by setting the `PYQGIS_STARTUP` environment variable to the path of an existing Python file.

This method is something you will probably rarely need, but worth mentioning here because it is one of the several ways to run Python code within QGIS and because this code will run before QGIS initialization is complete. This method is very useful for cleaning sys.path, which may have undesireable paths, or for isolating/loading the initial environ without requiring a virt env, e.g. homebrew or MacPorts installs on Mac.

### 1.1.2 The `startup.py` file

Every time QGIS starts, the user's Python home directory (usually: `.qgis2/python`) is searched for a file named `startup.py`, if that file exists, it is executed by the embedded Python interpreter.

## 1.2 Python コンソール

スクリプト処理をする上で、(QGIS に) 統合されている Python コンソールから多くの利点を得られるでしょう。これはメニューの **プラグイン** → *Python* **コンソール** から開くことができます。コンソールはモーダルではないユーティリティウィンドウに開きます:

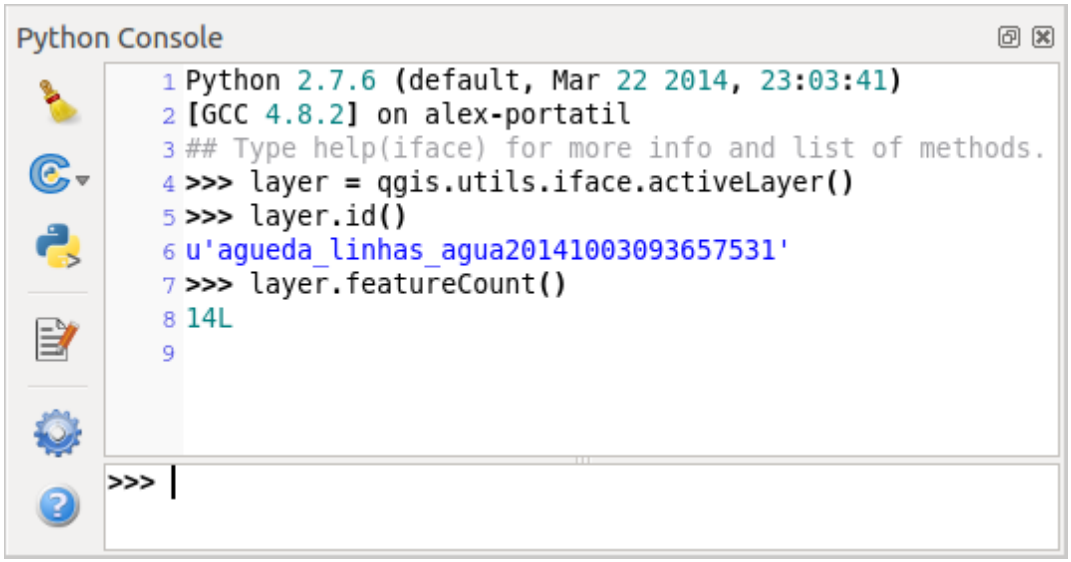


Figure 1.1: QGIS Python コンソール

上記のスクリーンショットはレイヤーのリストから現在選択中のレイヤーを取得して、ID や他の情報を表示してるところを見せていて、もしベクターレイヤーであれば、フューチャーの数を表示することができます。QGIS 環境とやりとりするには `QgisInterface` のインスタンスである `qgis.utils.iface` 変数を使います。このインターフェイスはマップキャンバス、メニュー、ツールバーやその他の QGIS アプリケーションのパーツにアクセスすることができます。

利便性を上げるためには、コンソールがスタートしたときに次の命令を入力してください (将来的には初期実行されるコマンドになるかもしれません):

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

このコンソールをたびたび使うなら、ショートカットを設定しておくとよいでしょう (メニューの **設定** → **ショートカットの構成**... から行えます)

## 1.3 Python プラグイン

QGIS はプラグインによる機能拡張が可能です。元々は C++でのみ可能でした。QGIS に Python サポートを追加したことで、Python でもプラグインを書く事ができるようになりました。C++プラグインよりもよりよい利点は簡単な配布 (プラットフォームごとのコンパイルする必要がありません) ができ、また簡単に開発ができます。

様々な機能をカバーする多くのプラグインは Python サポートが導入されてから書かれました。プラグインのインストーラは Python プラグインの取得、アップグレード、削除を簡単に行えます。様々なプラグインのソースが Python Plugin Repositories から見つけることができます。

Python でプラグインを作るのはとても簡単です。詳細は *plugins* を見てください。

## 1.4 Python アプリケーション

GIS データを処理するときは、繰り返し同じタスクを実行するのに簡単なスクリプトを書いて自動化することがたびたびあります。PyQGIS なら完璧に行えます — `qgis.core` モジュールを import すれば、初期化が行われて処理を行う準備が完了します。

もしくはいくつかの GIS の機能 — いくつかのデータの距離を測ったり、地図を PDF に変換したり、または他の機能など – を使ったインタラクティブなアプリケーションを作りたいのかもしれません。 `qgis.gui` モジュールは様々な GUI コンポーネントを追加することができ、とりわけマップキャンバスの widget はズームやパンや他のマップを制御するツールと一緒にアプリケーションに簡単に組み込むことができます。

### 1.4.1 Using PyQGIS in custom application

注意: `qgis.py` という名前をあなたのテストスクリプトで*使わないでください* — このスクリプトの名前がバインディングを隠蔽してしまって python で import できなくなるでしょう。

First of all you have to import qgis module, set QGIS path where to search for resources — database of projections, providers etc. When you set prefix path with second argument set as `True`, QGIS will initialize all paths with standard dir under the prefix directory. Calling `initQgis()` function is important to let QGIS search for the available providers.

```
from qgis.core import *

# supply path to where is your qgis installed
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# load providers
QgsApplication.initQgis()
```

これで QGIS API — レイヤーを読み込んだり、処理を行ったり、マップキャンバスと共に GUI を起動したり - を動かす事ができます。可能性は無限です :-)

When you are done with using QGIS library, call `exitQgis()` to make sure that everything is cleaned up (e.g. clear map layer registry and delete layers):

```
QgsApplication.exitQgis()
```

### 1.4.2 カスタムアプリケーションを実行する

QGIS のライブラリと Python モジュールが一般的な場所に置かれて無ければ、システムに適切な場所を伝える必要があるでしょう — そうしないと Python はエラーを吐きます:

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

これは `PYTHONPATH` という環境変数をセットすれば治ります。次のコマンドで `qgispath` の部分を実際に QGIS をインストールした場所に差し替えてください:

- Linux では: **export PYTHONPATH=/qgispath/share/qgis/python**
- Windows では: **set PYTHONPATH=c:\qgispath\python**

これで PyQGIS モジュールのパスがわかるようになりました。一方これらは `qgis_core` と ‘‘qgis_gui‘‘ライブラリに依存します (Python ライブラリはラッパーとして振る舞うだけです)。これらのライブラリのパスが OS で読み込めないものであれば、またもや import エラーが発生するでしょう (このメッセージはシステムにかなり依存していることを示します):

```
>>> import qgis.core
ImportError: libqgis_core.so.1.5.0: cannot open shared object file: No such file or directory
```

これを修正するには QGIS ライブラリが存在するディレクトリを動的リンカのパスに追加するをします:

- Linux では: **export LD_LIBRARY_PATH=/qgispath/lib**
- Windows では: **set PATH=C:\qgispath;%PATH%**

これらのコマンドはブートストラップスクリプトに入れておくことができます。PyQGIS を使ったカスタムアプリケーションを配布するには、これらの二つの方法が可能でしょう:

- QGIS を対象となるプラットフォームにインストールするのをユーザに要求します。アプリケーションのインストーラは QGIS ライブラリの標準的な場所を探すことができ、もし見つからなければユーザがパスをセットできるようにします。この手段はシンプルである利点がありますが、しかしながらユーザに多くの手順を要求します。
- アプリケーションと一緒に QGIS のパッケージを配布する方法です。アプリケーションのリリースにはいろいろやることがあるし、パッケージも大きくなりますが、ユーザを追加ソフトウェアをダウンロードをしてインストールする負荷から避けられるでしょう。

これらのモデルは組み合わせることができます - Windows と Mac OS X ではスタンドアローンアプリケーションとして配布をして、Linux では QGIS のインストールをユーザとユーザが使っているパッケージマネージャに任せるとか。

# Chapter 2

# Loading Projects

Sometimes you need to load an existing project from a plugin or (more often) when developing a stand-alone QGIS Python application (see: *Python* **アプリケーション**).

To load a project into the current QGIS aplication you need a `QgsProject instance()` object and call its `read()` method passing to it a `QFileInfo` object that contains the path from where the project will be loaded:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt4 classes you will use in this script as shown below:
from qgis.core import QgsProject
from PyQt4.QtCore import QFileInfo
# 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
u'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
print project.fileName
u'/home/user/projects/my_other_qgis_project.qgs'
```

In case you need to make some modifications to the project (for example add or remove some layers) and save your changes, you can call the `write()` method of your project instance. The `write()` method also accepts an optional `QFileInfo` that allows you to specify a path where the project will be saved:

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

Both `read()` and `write()` funtions return a boolean value that you can use to check if the operation was successful.

# Chapter 3

# レイヤのロード

データのレイヤをオープンしましょう.QGIS はベクタとラスタレイヤを認識できます. 加えてカスタムレイヤタイプを利用することもできますが, それについてここでは述べません.

## 3.1 ベクタレイヤ

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

```
layer = QgsVectorLayer(data_source, layer_name, provider_name)
if not layer.isValid():
  print "Layer failed to load!"
```

データソース識別子は文字列でそれぞれのデータプロバイダを表します. レイヤ名はレイヤリストウィジェットで使われます. レイヤが正常にロードされたかどうかをチェックすることは重要です. 正しくロードされていない場合は不正なレイヤインスタンスが返ります.

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

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

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step. The function returns the layer instance or *None* if the layer couldn't be loaded.

以下のリストはベクタデータプロバイダを使って様々なデータソースにアクセスする方法が記述されています.

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

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

- PostGIS database — data source is a string with all information needed to create a connection to PostgreSQL database. `QgsDataSourceURI` class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn't available.

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

  vlayer = QgsVectorLayer(uri.uri(), "layer_name_you_like", "postgres")
  ```

- CSV or other delimited text files — to open a file with a semicolon as a delimiter, with field "x" for x-coordinate and field "y" with y-coordinate you would use something like this

```
uri = "/some/path/file.csv?delimiter=%s&xField=%s&yField=%s" % (";", "x", "y")
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "delimitedtext")
```

  Note: from QGIS version 1.7 the provider string is structured as a URL, so the path must be prefixed with *file:///*. Also it allows WKT (well known text) formatted geometries as an alternative to "x" and "y" fields, and allows the coordinate reference system to be specified. For example

```
uri = "file:///some/path/file.csv?delimiter=%s&crs=epsg:4723&wktField=%s" % (";", "shape")
```

- GPX files — the "gpx" data provider reads tracks, routes and waypoints from gpx files. To open a file, the type (track/route/waypoint) needs to be specified as part of the url

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "gpx")
```

- SpatiaLite database — supported from QGIS v1.1. Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier

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

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

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

```
uri = "MySQL:dbname,host=localhost,port=3306,user=root,password=xxx|layername=my_table"
vlayer = QgsVectorLayer( uri, "my_table", "ogr" )
```

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

```
uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=union&version=1.0.0&re
vlayer = QgsVectorLayer("my_wfs_layer", "WFS")
```

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

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

## 3.2 ラスタレイヤ

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

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
```

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

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

```
iface.addRasterLayer("/path/to/raster/file.tif", "layer_name_you_like")
```

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step.

Raster layers can also be created from a WCS service.

```
layer_name = 'modis'
uri = QgsDataSourceURI()
uri.setParam('url', 'http://demo.mapserver.org/cgi-bin/wcs')
uri.setParam("identifier", layer_name)
rlayer = QgsRasterLayer(str(uri.encodedUri()), 'my_wcs_layer', 'wcs')
```

detailed URI settings can be found in provider documentation

Alternatively you can load a raster layer from WMS server. However currently it's not possible to access GetCapabilities response from API — you have to know what layers you want

```
urlWithParams = 'url=http://wms.jpl.nasa.gov/wms.cgi&layers=global_mosaic&styles=pseudo&format=ima
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print "Layer failed to load!"
```

## 3.3 マップレイヤレジストリ

もしあなたが開かれているレイヤを描画に利用したい場合はそれらをマップレイヤレジストリに追加することを忘れないで下さい. マップレイヤレジストリはレイヤのオーナーシップを取得して後でアプリケーションのいろいろな場所でユニーク ID を使ってアクセスできるようになります. レイヤがマップレイヤレジストリから削除された削除された時にそれも削除されます.

Adding a layer to the registry

```
QgsMapLayerRegistry.instance().addMapLayer(layer)
```

Layers are destroyed automatically on exit, however if you want to delete the layer explicitly, use

```
QgsMapLayerRegistry.instance().removeMapLayer(layer_id)
```

For a list of loaded layers and layer ids, use

```
QgsMapLayerRegistry.instance().mapLayers()
```

**TODO:** More about map layer registry?

# Chapter 4

# ラスターレイヤを使う

このセクションではラスタレイヤに対して行える様々な操作について紹介していきます.

## 4.1 レイヤについて

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

```
rlayer.width(), rlayer.height()
(812, 301)
rlayer.extent()
<qgis._core.QgsRectangle object at 0x000000000F8A2048>
rlayer.extent().toString()
u'12.095833,48.552777 : 18.863888,51.056944'
rlayer.rasterType()
2  # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
rlayer.bandCount()
3
rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
rlayer.hasPyramids()
False
```

## 4.2 Drawing Style

When a raster layer is loaded, it gets a default drawing style based on its type. It can be altered either in raster layer properties or programmatically. The following drawing styles exist:

| Index | Constant: QgsRasterLater.X | Comment |
|---|---|---|
| 1 | SingleBandGray | Single band image drawn as a range of gray colors |
| 2 | SingleBandPseudoColor | Single band image drawn using a pseudocolor algorithm |
| 3 | PalettedColor | “Palette” image drawn using color table |
| 4 | PalettedSingleBandGray | “Palette” layer drawn in gray scale |
| 5 | PalettedSingleBandPseudoColor | “Palette” layer drawn using a pseudocolor algorithm |
| 7 | MultiBandSingleBandGray | Layer containing 2 or more bands, but a single band drawn as a range of gray colors |
| 8 | MultiBandSingleBandPseudoColor | Layer containing 2 or more bands, but a single band drawn using a pseudocolor algorithm |
| 9 | MultiBandColor | Layer containing 2 or more bands, mapped to RGB color space. |

To query the current drawing style:

```
rlayer.renderer().type()
u'singlebandpseudocolor'
```

単バンドラスタレイヤはグレースケール (低い値=黒, 高い値=白) でも, 単バンドの値に色を割り当てたシュードカラーでも表示できます. また, 単バンドラスタはカラーマップでも表示できます. マルチバンドラスタは基本的に RGB カラーが割り当てて表示されますが, いずれかのバンドをグレースケールやシュードカラーで表示することもできます.

続いてのセクションではどのようにレイヤの表示方法を探したり変更するのかを説明していきます. 設定変更後にマップキャンバスの表示も更新をしたい場合は, **レイヤの更新** を参考にしてください.

**TODO:** *特定の値の強調, 透過 (No Data), ユーザー定義の最大値・最小値, バンド統計

### 4.2.1 単バンドラスタ

They are rendered in gray colors by default. To change the drawing style to pseudocolor:

```
# Check the renderer
rlayer.renderer().type()
u'singlebandgray'
rlayer.setDrawingStyle("SingleBandPseudoColor")
# The renderer is now changed
rlayer.renderer().type()
u'singlebandpseudocolor'
# Set a color ramp hader function
shader_func = QgsColorRampShader()
rlayer.renderer().shader().setRasterShaderFunction(shader_func)
```

The `PseudoColorShader` is a basic shader that highlights low values in blue and high values in red. There is also `ColorRampShader` which maps the colors as specified by its color map. It has three modes of interpolation of values:

- 線形 (補間): カラーマップで色を指定した値とその間を線形補間により色を割りてます.
- (離散的): カラーマップで指定された値及びそれ以上の値を同じ色に設定します.
- (厳密): 色の補間を行わず, カラーマップで指定された値のみを表示します.

To set an interpolated color ramp shader ranging from green to yellow color (for pixel values from 0 to 255):

```
rlayer.renderer().shader().setRasterShaderFunction(QgsColorRampShader())
lst = [QgsColorRampShader.ColorRampItem(0, QColor(0, 255, 0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255, 255 ,0))]
fcn = rlayer.renderer().shader().rasterShaderFunction()
fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
fcn.setColorRampItemList(lst)
```

To return back to default gray levels, use:

```
rlayer.setDrawingStyle('SingleBandGray')
```

### 4.2.2 マルチバンドラスタ

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

```
rlayer.setDrawingStyle('MultiBandColor')
rlayer.renderer().setGreenBand(1)
rlayer.setRedBand(2)
```

## 4.3 レイヤの更新

If you do change layer symbology and would like ensure that the changes are immediately visible to the user, call these methods

```
if hasattr(layer, "setCacheImage"):
  layer.setCacheImage(None)
layer.triggerRepaint()
```

一つ目の方法は, キャッシュ表示をオンにした際に表示レイヤのキャッシュ画像を削除するやり方です. この機能は QGIS 1.4 以降で使用可能になりました.

二つ目の方法は更新したいマップキャンバス上のレイヤを指定して除去するやり方です.

With WMS raster layers, these commands do not work. In this case, you have to do it explicitly

```
layer.dataProvider().reloadData()
layer.triggerRepaint()
```

In case you have changed layer symbology (see sections about raster and vector layers on how to do that), you might want to force QGIS to update the layer symbology in the layer list (legend) widget. This can be done as follows (`iface` is an instance of `QgisInterface`)

```
iface.legendInterface().refreshLayerSymbology(layer)
```

## 4.4 値の検索

To do a query on value of bands of raster layer at some specified point

```
ident = rlayer.dataProvider().identify(QgsPoint(15.30, 40.98), \
  QgsRaster.IdentifyFormatValue)
if ident.isValid():
  print ident.results()
```

この場合の `results` メソッドは、キーとしてバンドインデックスを持ち、値としてバンド値を持つ辞書型を返します。

```
{1: 17, 2: 220}
```

# Chapter 5

# ベクターレイヤを使う

このセクションではベクタレイヤに対して行える様々な操作について紹介していきます.

## 5.1 Retrieving informations about attributes

You can retrieve informations about the fields associated with a vector layer by calling `pendingFields()` on a `QgsVectorLayer` instance:

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

## 5.2 フィーチャの選択

In QGIS desktop, features can be selected in different ways, the user can click on a feature, draw a rectangle on the map canvas or use an expression filter. Selected fatures are normally higlighted in a different color (default is yellow) to draw user's attention on the selection. Sometimes can be useful to programmatically select features or to change the default color.

To change the selection color you can use `setSelectionColor()` method of `QgsMapCanvas` as shown in the following example:

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

To add add features to the selected features list for a given layer, you can call `setSelectedFeatures()` passing to it the list of features IDs:

```
# Get the active layer (must be a vector layer)
layer = iface.activeLayer()
# Get the first feature from the layer
feature = layer.getFeatures().next()
# Add this features to the selected list
layer.setSelectedFeatures([feature.id()])
```

選択を解除するため、空のリストを通ります。

```
layer.setSelectedFeatures([])
```

## 5.3 ベクターレイヤの反復処理

ベクターレイヤのフィーチャへの反復処理はもっとも頻繁に行う処理の一つです。次の例はこの処理を行う基本的なコードで、各フィーチャのいくつかの情報を表示します。 `layer` は QGSVectorLayer オブジェクトとしています。

```
iter = layer.getFeatures()
for feature in iter:
    # retrieve every feature with its geometry and attributes
    # fetch geometry
    geom = feature.geometry()
    print "Feature ID %d: " % feature.id()

    # show some information about the feature
    if geom.type() == QGis.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == QGis.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == QGis.Polygon:
        x = geom.asPolygon()
        numPts = 0
        for ring in x:
        numPts += len(ring)
        print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
        print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

    # attrs is a list. It contains all the attribute values of this feature
    print attrs
```

### 5.3.1 属性のアクセス

属性は、それらの名称によって参照されます。

```
print feature['name']
```

あるいは、属性はインデックスに参照されます。これは名称を使うよりもやや高速です。たとえば、新しい属性を取得するためです:

```
print feature[0]
```

### 5.3.2 選択されたフィーチャへの反復処理

地物を選択する必要のみある場合、 ベクタレイヤから :func: *selectedFeatures* メソッドを使用できます。

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

Another option is the Processing `features()` method:

```
import processing
features = processing.features(layer)
```

```
for feature in features:
    # do whatever you need with the feature
```

By default, this will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise. Note that this behavior can be changed in the Processing options to ignore selections.

### 5.3.3 一部のフィーチャへの反復処理

もし所定の範囲内に含まれフィーチャのように、レイヤ中の所定のフィーチャにのみ処理を行いたい場合、`QgsFeatureRequest` オブジェクトを `getFeatures()` に加えます。下記が例になります。

```
request = QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for feature in layer.getFeatures(request):
    # do whatever you need with the feature
```

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the example above, you can build an `QgsExpression` object and pass it to the `QgsFeatureRequest` constructor. Here's an example

```
# The expression will filter the features where the field "location_name" contains
# the word "Lake" (case insensitive)
exp = QgsExpression('location_name ILIKE \'%Lake%\'')
request = QgsFeatureRequest(exp)
```

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

```
# Only return selected fields
request.setSubsetOfAttributes([0,2])
# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.pendingFields())
# Don't return geometry objects
request.setFlags(QgsFeatureRequest.NoGeometry)
```

---

**ちなみに:** If you only need a subset of the attributes or you don't need the geometry informations, you can significantly increase the **speed** of the features request by using `QgsFeatureRequest.NoGeometry` flag or specifying a subset of attributes (possibly empty) like shown in the example above.

---

## 5.4 ベクターレイヤの修正

Most vector data providers support editing of layer data. Sometimes they support just a subset of possible editing actions. Use the `capabilities()` function to find out what set of functionality is supported

```
caps = layer.dataProvider().capabilities()
```

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

---

**ノート:** If you are working inside QGIS (either from the console or from a plugin), it might be necessary to force a redraw of the map canvas in order to see the changes you've done to the geometry, to the style or to the attributes:

```
# If caching is enabled, a simple canvas refresh might not be sufficient
# to trigger a redraw and you must clear the cached image for the layer
if iface.mapCanvas().isCachingEnabled():
    layer.setCacheImage(None)
else:
    iface.mapCanvas().refresh()
```

### 5.4.1 フィーチャの追加

Create some `QgsFeature` instances and pass a list of them to provider's `addFeatures()` method. It will return two values: result (true/false) and list of added features (their ID is set by the data store)

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature()
    feat.addAttribute(0, 'hello')
    feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
    (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

### 5.4.2 フィーチャの削除

フィーチャを削除するには、フィーチャの ID の配列を渡すだけです:

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

### 5.4.3 フィーチャの修正

フィーチャのジオメトリの変更も属性の変更もどちらも可能です。次のサンプルは最初にインデックス 0 と 1 の属性の値を変更し、その後にフィーチャのジオメトリを変更しています

```
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.fromPoint(QgsPoint(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

---

**ちなみに:** If you only need to change geometries, you might consider using the `QgsVectorLayerEditUtils` which provides some of useful methods to edit geometries (translate, insert or move vertex etc.)

---

### 5.4.4 フィールドの追加または削除

フィールド (属性) を追加するには、フィールドの定義の配列を指定する必要があります。フィールドを削除するにはフィールドのインデックスの配列を渡すだけです

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

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

データプロバイダのフィールドを追加または削除した後、レイヤのフィールドは、変更が自動的に反映されていないため、更新する必要があります。

```
layer.updateFields()
```

## 5.5 ベクターレイヤを編集バッファで修正する.

QGIS アプリケーションでベクターを編集するには、個々のレイヤを編集モードにしてから編集を行って最後に変更をコミット (もしくはロールバック) します。全ての変更はそれらをコミットするまでは書き込まれません — これらはメモリ上の編集バッファに居続けます。これらの機能はプログラムで扱うことができます　— これはデータプロバイダを直接使う方法を補完するベクターレイヤを編集する別の方法です。ベクターレイヤの編集機能をもった GUI ツールを提供する際にこのオプションを使えば、ユーザにコミット/ロールバックをするのを決めさせられ、また undo/redo のような使い方をさせることができます。変更をコミットする時に、編集バッファの全ての変更はデータプロバイダに保存されます。

To find out whether a layer is in editing mode, use `isEditing()` — the editing functions work only when the editing mode is turned on. Usage of editing functions

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

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

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

適切に undo/redo が動くようにするには、上記で言及しているコマンドを undo コマンドでラップする必要があります。(もし undo/redo を気にしないで、逐一変更を保存するのであれば、 **データプロバイダでの編集** で簡単に実現できるでしょう。) undo 機能はこのように使います

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

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

if problem_occurred:
    layer.destroyEditCommand()
   return

# ... more editing ...

layer.endEditCommand()
```

`beginEndCommand()` は内部的に "アクティブな" コマンドを作成して、この後に続くベクターレイヤの変更を記録し続けます。　`endEditCommand()` を呼び出すことで undo スタックにコマンドがプッシュされ、ユーザが GUI からコマンドの undo/redo が可能になります。変更をしている途中でなにか問題が発生した場合は、 `destroyEditCommand()` メソッドでコマンドを削除してコマンドがアクティブであった時に行った全ての変更をロールバックするでしょう。

To start editing mode, there is `startEditing()` method, to stop editing there are `commitChanges()` and `rollback()` — however normally you should not need these methods and leave this functionality to be triggered by the user.

## 5.6 空間インデックスを使う

空間インデックスは、もしあなたが頻繁にベクターレイヤーに問い合わせをする必要がある場合、あなたのコードのパフォーマンスを劇的に改善することが出来ます。例えば、あなたが、補完値の計算に使用するために、与えられた場所に近接する 10 点をポイントレイヤーから求める必要がある、補完アルゴリズムを書いた場合を想像してください。空間インデックスが無い場合、QGIS がこれらの 10 点を求めるための

ただ一つの方法は、すべての点から指定の場所への距離を計算し、それらの距離を比較することです。これは、特に、いくつかの場所を求めるために繰り返す必要がある場合に、非常に時間のかかるタスクとなります。もし空間インデックスがレイヤに作成されている場合、処理はもっと効率的になります。

空間インデックスの無いレイヤは、電話番号が順番に並んでいない、もしくはインデックスの無い電話帳と思ってください。所定の人の電話番号を見つける唯一の方法は、巻頭からその番号を見つけるまで読むだけです。

Spatial indexes are not created by default for a QGIS vector layer, but you can create them easily. This is what you have to do.

1. 空間インデックスを作成する — 以下のコードは空のインデックスを作成する

   ```
   index = QgsSpatialIndex()
   ```

2. add features to index — index takes `QgsFeature` object and adds it to the internal data structure. You can create the object manually or use one from previous call to provider's `nextFeature()`

   ```
   index.insertFeature(feat)
   ```

3. 空間インデックスに何かしらの値が入れられると検索ができるようになります

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

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

## 5.7 ベクターレイヤの作成

`QgsVectorFileWriter` クラスを使ってベクターレイヤファイルを書き出す事ができます。これは OGR がサポートするいかなるベクターファイル (shapefiles, GeoJSON, KML そしてその他) をサポートしています。

ベクターレイヤをエクスポートする方法は二つあります:

- `QgsVectorLayer` インスタンスから

  ```
  error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_shapes.shp", "CP1250", None, "ESRI

  if error == QgsVectorFileWriter.NoError:
      print "success!"

  error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSO
  if error == QgsVectorFileWriter.NoError:
      print "success again!"
  ```

  The third parameter specifies output text encoding. Only some drivers need this for correct operation - shapefiles are one of those — however in case you are not using international characters you do not have to care much about the encoding. The fourth parameter that we left as `None` may specify destination CRS — if a valid instance of `QgsCoordinateReferenceSystem` is passed, the layer is transformed to that CRS.

  For valid driver names please consult the supported formats by OGR — you should pass the value in the "Code" column as the driver name. Optionally you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes — look into the documentation for full syntax.

- フィーチャから直接

  ```
  # define fields for feature attributes. A list of QgsField objects is needed
  fields = [QgsField("first", QVariant.Int),
            QgsField("second", QVariant.String)]
  ```

```
# create an instance of vector file writer, which will create the vector file.
# Arguments:
# 1. path to new file (will fail if exists already)
# 2. encoding of the attributes
# 3. field map
# 4. geometry type - from WKBTYPE enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Sh

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk (optional)
del writer
```

## 5.8 メモリープロバイダー

メモリープロバイダーはプラグインやサードパーティアプリケーション開発者に主に使われるでしょう。これはディスクにデータを保存せず、開発者がテンポラリなレイヤーの高速なバックエンドとして使えるようになります。

プロバイダは文字列と int と double をサポートします。

メモリープロバイダーは空間インデックスもサポートしていて、プロバイダーの `createSpatialIndex()` を呼ぶことで有効になります。一度空間インデックスを作成したら小さい領域内でフィーチャの iterate が高速にできるようになります (これ以降は全てのフィーチャを順にたどる必要がなくなり、指定した短形内で収まります)。

メモリープロバイダーは `QgsVectorLayer` のコンストラクタに `"memory"` をプロバイダーの文字列として与えると作成されます。

コンストラクタはレイヤーのジオメトリの種類に指定した URL を与えることができます。この種類は次のものです: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, `"MultiPolygon"`.

URI ではメモリープロバイダーの座標参照系、属性フィールド、インデックスを指定することが出来ます。構文は、

**crs=definition** 座標参照系を指定し、この定義は `QgsCoordinateReferenceSystem.createFromString()` で受け付ける事ができるどんな値でも置くことができます。

**index=yes** プロバイダーが空間インデックスを使うことを指定します。

**field=name:type(length,precision)** レイヤーの属性を指定します。属性は名前を持ち、オプションとして種類 (integer, double, string)、長さと正確性を持ちます。複数のフィールドの定義を置くことになるでしょう。

次のサンプルは全てのこれらのオプションを含んだ URL です:

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

次のサンプルコードはメモリープロバイダーを作成してデータ投入をしている様子です:

```
# 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.fromPoint(QgsPoint(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()
```

最後にやったことを全て確認していきましょう:

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

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

## 5.9 ベクタレイヤーの外観 (シンボロジ)

ベクタレイヤーがレンダリングされるとき、データの外観はレイヤーによって関連付けられた **レンダラー** と **シンボル** によって決定されます。シンボルはフィーチャの仮想的な表現を描画するクラスで、レンダラーはシンボルが個々のフィーチャで使われるかを決定します。

指定したレイヤのレンダラーは下記のように得ることが出来ます

```
renderer = layer.rendererV2()
```

この参照を利用して、少しだけ探索してみましょう:

```
print "Type:", rendererV2.type()
```

次の表は QGIS コアライブラリに存在するいくつかのよく知られたレンダラーです:

| タイプ | クラス | 詳細 |
|---|---|---|
| singleSymbol | QgsSingleSymbolRendererV2 | 全てのフィーチャを同じシンボルでレンダーします |
| categorizedSymbol | QgsCategorizedSymbolRendererV2 | カテゴリごとに違うシンボルを使ってフィーチャをレンダーします |
| graduatedSymbol | QgsGraduatedSymbolRendererV2 | それぞれの範囲の値によって違うシンボルを使ってフィーチャをレンダーします |

カスタムレンダラーのタイプになることもあるので、上記のタイプになるとは思い込まないでください。QgsRendererV2Registry シングルトンを検索して現在利用可能なレンダラーを見つけることもできます。

```
QgsRendererV2Registry.instance().renderersList()
# Prints:
[u'singleSymbol',
u'categorizedSymbol',
u'graduatedSymbol',
u'RuleRenderer',
u'pointDisplacement',
u'invertedPolygonRenderer',
u'heatmapRenderer']
```

レンダラーの中身をテキストフォームにダンプすることできます — デバッグ時に役に立つでしょう:

```
print rendererV2.dump()
```

### 5.9.1 単一シンボルレンダラ

レンダリングが使っているシンボルは `symbol()` メソッドで取得することができ、 `setSymbol()` メソッドで変更することができます (C++開発者へメモ: レンダラーはシンボルのオーナーシップをとります)。

You can change the symbol used by a particular vector layer by calling `setSymbol()` passing an instance of the appropriate symbol instance. Symbols for *point*, *line* and *polygon* layers can be created by calling the `createSimple()` function of the corresponding classes `QgsMarkerSymbolV2`, `QgsLineSymbolV2` and `QgsFillSymbolV2`.

The dictionary passed to `createSimple()` sets the style properties of the symbol.

For example you can change the symbol used by a particular **point** layer by calling `setSymbol()` passing an instance of a `QgsMarkerSymbolV2` as in the following code example:

```
symbol = QgsMarkerSymbolV2.createSimple({'name': 'square', 'color': 'red'})
layer.rendererV2().setSymbol(symbol)
```

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

- *circle‘*
- `square`
- `rectangle`
- *diamond‘*
- `pentagon`
- `triangle`
- `equilateral_triangle`
- `star`
- `regular_star`
- `arrow`
- *filled_arrowhead‘*

### 5.9.2 カテゴライズドシンボルレンダラ

分類するのに使われる属性名を検索したりセットしたりすることができます: `classAttribute()` メソッドと `setClassAttribute()` メソッドを使います。

カテゴリの配列を取得するには

```
for cat in rendererV2.categories():
    print "%s: %s :: %s" % (cat.value().toString(), cat.label(), str(cat.symbol()))
```

value() はカテゴリを区別にするのに使う値で、 label() はカテゴリの詳細に使われるテキストで、symbol() メソッドは割り当てられているシンボルを返します。

レンダラはたいていオリジナルのシンボルと識別をするためにカラーランプを保持しています: sourceColorRamp() メソッドと sourceSymbol() メソッドから呼び出せます。

### 5.9.3 階調シンボルレンダラ

このレンダラは先ほど暑かったカテゴリ分けシンボルのレンダラととても似ていますが、クラスごとの一つの属性値の代わりに領域の値として動作し、そのため数字の属性のみ使うことができます。

レンダラで使われている領域の多くの情報を見つけるには

```
for ran in rendererV2.ranges():
    print "%f - %f: %s %s" % (
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
        str(ran.symbol())
      )
```

属性名の分類を調べるために classAttribute() をまた使うことができ、 sourceSymbol() メソッドと sourceColorRamp() メソッドも使うことができます。さらに作成された領域の測定する mode() メソッドもあります: 等間隔や変位値、その他のメソッドと一緒に使います。

もし連続値シンボルレンダラを作ろうとしているのであれば次のスニペットの例で書かれているようにします (これはシンプルな二つのクラスを作成するものを取り上げています):

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(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 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRendererV2(myRenderer)
QgsMapLayerRegistry.instance().addMapLayer(myVectorLayer)
```

### 5.9.4 シンボルの操作

シンボルを表現するには、 QgsSymbolV2 ベースクラス由来の三つの派生クラスを使います:

- QgsMarkerSymbolV2 - ポイントのフィーチャ用
- QgsLineSymbolV2 - ラインのフィーチャ用
- QgsFillSymbolV2 - ポリゴンのフィーチャ用

**全てのシンボルは一つ以上のシンボルレイヤーから構成されます** (QgsSymbolLayerV2 の派生クラスです)。シンボルレイヤーは実際にレンダリングをして、シンボルクラス自信はシンボルレイヤのコンテナを提供するだけです。

(例えばレンダラから) シンボルのインスタンスを持っていればそれの中身を調べる事ができます: type() メソッドはそれ自身がマーカか、ラインか、シンボルで満たさたものかを返します。 dump() メソッドはシンボルの簡単な説明を返します。シンボルレイヤーの配列を取得するにはこのようにします:

```
for i in xrange(symbol.symbolLayerCount()):
    lyr = symbol.symbolLayer(i)
    print "%d: %s" % (i, lyr.layerType())
```

シンボルが使っている色を得るには color() メソッドを使い、 setColor() でシンボルの色を変えます。マーカーシンボルは他にもシンボルのサイズと回転角をそれぞれ size() メソッドと angle() メソッドで取得することができ、ラインシンボルは width() メソッドでラインの幅を返します。

サイズと幅は標準でミリメートルが使われ、角度は 度 が使われます。

#### シンボルレイヤの操作

前に述べたようにシンボルレイヤ (QgsSymbolLayerV2 のサブクラスです) はフィーチャの外観を決定します。一般的に使われるいくつかの基本となるシンボルレイヤのクラスがあります。これは新しいシンボルレイヤの種類を実装を可能とし、それによってフィーチャがどのようにレンダーされるかを任意にカスタマイズできます。 layerType() メソッドはシンボルレイヤクラスの一意に識別します — 基本クラスは標準で SimpleMarker 、 SimpleLine 、 SimpleFill がシンボルレイヤのタイプとなります。

次のようにシンボルレイヤクラスを与えてシンプルレイヤを作成して、シンボルレイヤのタイプの完全なリストを取得することができます。

```
from qgis.core import QgsSymbolLayerV2Registry
myRegistry = QgsSymbolLayerV2Registry.instance()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbolV2.Marker):
    print item
```

出力

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

QgsSymbolLayerV2Registry クラスは利用可能な全てのシンボルレイヤタイプのデータベースを管理しています。

シンボルレイヤのデータにアクセスするには、 properties() メソッドを使い、これは表現方法を決定しているプロパティの辞書のキー値を返します。それぞれのシンボルレイヤタイプはそれが使っている特定のプロパティの集合を持っています。さらに、共通して使えるメソッドとして color(), size(), angle(), width() がそれぞれセッターと対応して存在します。もちろん size と angle はマーカーシンボルレイヤだけで利用可能で、 width はラインシンボルレイヤだけで利用可能です。

### カスタムシンボルレイヤタイプの作成

あなたがデータをどうレンダーするかをカスタマイズしたいと考えているとします。あなたはあなたが思うままにフィーチャを描画する独自のシンボルレイヤクラスを作ることができます。次の例は指定した半径で赤い円を描画するマーカを示しています:

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

  def __init__(self, radius=4.0):
      QgsMarkerSymbolLayerV2.__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)
```

`layerType()` メソッドはシンボルレイヤーの名前を決定し、全てのシンボルレイヤーの中で一意になります。プロパティは属性の持続として使われます。`clone()` メソッドは全ての全く同じ属性を含んだシンボルレイヤーのコピーを返さなくてはなりません。最後にレンダリングのメソッドについて: `startRender()` はフィーチャが最初にレンダリングされる前に呼び出され、 `stopRender()` はレンダリングが終わったら呼び出されます。そして `renderPoint()` メソッドでレンダリングを行います。ポイントの座標は出力対象の座標に常に変換されます。

ポリラインとポリゴンではレンダリングのメソッドが違うだけです: (ポリラインでは) それぞれのラインの配列を受け取る `renderPolyline()` を使います。 `renderPolygon()` は最初のパラメータを外輪としたポイントのリストと、２つ目のパラメータに内輪 (もしくは None) のリストを受け取ります。

普通はユーザに外観をカスタマイズさせるためにシンボルレイヤータイプの属性を設定する GUI を追加すると使いやすくなります: 上記の例であればユーザは円の半径をセットできます。次のコードは widget の実装となります:

```
class FooSymbolLayerWidget(QgsSymbolLayerV2Widget):
    def __init__(self, parent=None):
        QgsSymbolLayerV2Widget.__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()"))
```

この widget はシンボルプロパティのダイアログに組み込むことができます。シンボルプロパティのダイアログでシンボルレイヤータイプを選択したときにこれはシンボルレイヤーのインスタンスとシンボルレイヤー widget のインスタンスを作成します。そして widget をシンボルレイヤーを割り当てるために `setSymbolLayer()` メソッドを呼び出します。このメソッドで widget がシンボルレイヤーの属性を反映するよう UI を更新します。 `symbolLayer()` 関数はシンボルが使ってるプロパティダイアログがシンボルレイヤーを再度探すのに使われます。

いかなる属性の変更時でも、プロパティダイアログにシンボルプレビューを更新させるために widget は `changed()` シグナルを発生します。

私達は最後につなげるところだけまだ扱っていません: QGIS にこれらの新しいクラスを知らせる方法です。これはレジストリにシンボルレイヤーを追加すれば完了です。レジストリに追加しなくてもシンボルレイヤーを使うことはできますが、いくつかの機能が動かないでしょう: 例えばカスタムシンボルレイヤーを使ってプロジェクトファイルを読み込んだり、GUI でレイヤーの属性を編集できないなど。

私達はシンボルレイヤーのメタデータを作る必要があります

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

  def __init__(self):
    QgsSymbolLayerV2AbstractMetadata.__init__(self, "FooMarker", QgsSymbolV2.Marker)

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

  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

QgsSymbolLayerV2Registry.instance().addSymbolLayerType(FooSymbolLayerMetadata())
```

レイヤータイプ (レイヤーが返すのと同じもの) とシンボルタイプ (marker/line/fill) を親クラスのコンストラクタに渡します。 `createSymbolLayer()` は辞書の引数の *props* で指定した属性をもつシンボルレイヤーのインスタンスを作成をしてくれます。(キー値は QString のインスタンスで、決して “str” のオブジェクトではないのに気をつけましょう) そして `createSymbolLayerWidget()` メソッドはこのシンボルレイヤータイプの設定 widget を返します。

最後にこのシンボルレイヤーをレジストリに追加します — これで完了です。

### 5.9.5 カスタムレンダラの作成

もしシンボルがフィーチャのレンダリングをどう行うかをカスタマイズしたいのであれば、新しいレンダラーの実装を作ると便利かもしれません。いくつかのユースケースとしてこんなことをしたいのかもしれません: フィールドの組み合わせからシンボルを決定する、現在の縮尺に合わせてシンボルのサイズを変更するなどなど。

次のコードは二つのマーカーシンボルを作成して全てのフィーチャからランダムに一つ選ぶ簡単なカスタムレンダラです

```
import random

class RandomRenderer(QgsFeatureRendererV2):
  def __init__(self, syms=None):
    QgsFeatureRendererV2.__init__(self, "RandomRenderer")
    self.syms = syms if syms else [QgsSymbolV2.defaultSymbol(QGis.Point), QgsSymbolV2.defaultSymbo

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

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

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

  def usedAttributes(self):
    return []

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

親クラスの QgsFeatureRendererV2 のコンストラクタはレンダラの名前 (レンダラの中で一意になる必要があります) が必要です。 symbolForFeature() メソッドは個々のフィーチャでどのシンボルが使われるかを一つ決定します。 startRender() と stopRender() それぞれシンボルのレンダリングの初期化/終了を処理します。 usedAttributes() メソッドはレンダラが与えられるのを期待するフィールド名のリストを返すことができます。最後に clone() 関数はレンダラーのコピーを返すでしょう。

シンボルレイヤー同様、レンダラの設定を GUI からいじることができます。これは QgsRendererV2Widget の派生クラスとなります。次のサンプルコードではユーザが最初のシンボルのシンボルをセットするボタンを作成しています (訳注: サンプルを見ると色を変更しているので原文が間違っていると思われる)

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2("Color 1")
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

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

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

コンストラクタはアクティブなレイヤー (QgsVectorLayer) とグローバルなスタイル (QgsStyleV2) と現在のレンダラのインスタンスを受け取ります。もしレンダラが無かったり、レンダラが違う種類のもの

だったら、コンストラクタは新しいレンダラに差し替えるか、そうでなければ現在のレンダラー (必要な種類を持つでしょう) を使います。widget の中身はレンダラーの現在の状態を表示するよう更新されます。レンダラダイアログが受け入れられたときに、現在のレンダラを取得するために widget の `renderer()` メソッドが呼び出されます。

最後のちょっとした作業はレンダラのメタデータとレジストリへの登録で、これらをしないとレンダラのレイヤーの読み込みは動かなく、ユーザはレンダラのリストから選択することができないでしょう。では、私達の RandomRenderer の例を終わらせましょう

```
class RandomRendererMetadata(QgsRendererV2AbstractMetadata):
  def __init__(self):
    QgsRendererV2AbstractMetadata.__init__(self, "RandomRenderer", "Random renderer")

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

QgsRendererV2Registry.instance().addRenderer(RandomRendererMetadata())
```

シンボルレイヤーと同様に、abstract metadata のコンストラクタはレンダラの名前を受け取るのを期待して、この名前はユーザに見え、レンダラのアイコンの追加の名前となります。`createRenderer()` メソッドには `QDomElement` のインスタンスを渡してレンダラの状態を DOM ツリーから復元するのに使います。`createRendererWidget()` メソッドは設定の widget を作成します。これは必ず存在する必要はなく、もしレンダラが GUI からいじらないのであれば *None* を返すことができます。

レンダラにアイコンを関連付けるには `QgsRendererV2AbstractMetadata` のコンストラクタの三番目の引数 (オプション) に指定することができます — RandomRendererMetadata の `__init__()` 関数の中の基本クラスのコンストラクタはこうなります

```
QgsRendererV2AbstractMetadata.__init__(self,
       "RandomRenderer",
       "Random renderer",
       QIcon(QPixmap("RandomRendererIcon.png", "png")))
```

アイコンはあとからメタデータクラスの `setIcon()` を使って関連付けることもできます。アイコンはファイルから読み込むこと (上記を参考) も Qt のリソース から読み込むこともできます (PyQt4 はパイソン向けの .qrc コンパイラを含んでいます)。

## 5.10 より詳しいトピック

**TODO:** creating/modifying symbols working with style (`QgsStyleV2`) working with color ramps (`QgsVectorColorRampV2`) rule-based renderer (see this blogpost) exploring symbol layer and renderer registries

# Chapter 6

# ジオメトリの操作

空間的な特徴を表すポイント、ライン、ポリゴンは一般的にジオメトリと呼ばれています。QGIS では `QgsGeometry` クラスで代表されます。すべてのジオメトリタイプは JTS discussion page でよく示されています。

時には 1 つのジオメトリは実際に単純な（シングルパート）ジオメトリの集合です。このような幾何学的形状は、マルチパートジオメトリと呼ばれています。単純にジオメトリのちょうど 1 種類が含まれている場合は、マルチポイント、マルチラインまたはマルチポリゴンと呼んでいます。例えば、複数の島からなる国は、マルチポリゴンのように表すことができます。

ジオメトリの座標値はどの座標参照系 (CRS) も利用できます。レイヤーからフィーチャを持ってきたときに、ジオメトリの座標値はレイヤーの CRS のものを持つでしょう。

## 6.1 ジオメトリの構成

ジオメトリの作成にはいくつかのオプションがあります。

- from coordinates

  ```
  gPnt = QgsGeometry.fromPoint(QgsPoint(1,1))
  gLine = QgsGeometry.fromPolyline([QgsPoint(1, 1), QgsPoint(2, 2)])
  gPolygon = QgsGeometry.fromPolygon([[QgsPoint(1, 1), QgsPoint(2, 2), QgsPoint(2, 1)]])
  ```

  座標値は `QgsPoint` クラスを使って与えられます。

  ポリライン (ラインストリング) はポイントのリストで表現されます。ポリゴンは線形の輪 (すなわち閉じたラインストリング) のリストで表現されます。最初の輪は外輪 (境界) で、オプションとして続く輪がポリゴン内の穴となります。

  マルチパートジオメトリはさらに上のレベルです: マルチポイントはポイントのリストで、マルチラインストリングはラインストリングのリストで、マルチポリゴンはポリゴンのリストです。

- from well-known text (WKT)

  ```
  gem = QgsGeometry.fromWkt("POINT(3 4)")
  ```

- from well-known binary (WKB)

  ```
  g = QgsGeometry()
  g.setWkbAndOwnership(wkb, len(wkb))
  ```

## 6.2 ジオメトリにアクセス

First, you should find out geometry type, `wkbType()` method is the one to use — it returns a value from `QGis.WkbType` enumeration

```
>>> gPnt.wkbType() == QGis.WKBPoint
True
>>> gLine.wkbType() == QGis.WKBLineString
True
>>> gPolygon.wkbType() == QGis.WKBPolygon
True
>>> gPolygon.wkbType() == QGis.WKBMultiPolygon
False
```

As an alternative, one can use `type()` method which returns a value from `QGis.GeometryType` enumeration. There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

全てのベクタータイプにジオメトリから情報を展開するのに使えるアクセサ関数があります。アクセサはこのように使います:

```
>>> gPnt.asPoint()
(1, 1)
>>> gLine.asPolyline()
[(1, 1), (2, 2)]
>>> gPolygon.asPolygon()
[[(1, 1), (2, 2), (2, 1), (1, 1)]]
```

注意: このタプル (x, y) は本当のタプルではなく、これらは `QgsPoint` のオブジェクトで、この値は `x()` メソッド及び `y()` メソッドでアクセスできるようになっています。

マルチパートジオメトリ同士で似たようなアクセサ関数があります: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()` です。

## 6.3 ジオメトリの述語と操作

QGIS はジオメトリ述部 (`contains()`, `intersects()`, ...) や操作設定 (`union()`, `difference()`, ...) のような上級のジオメトリ操作で GEOS ライブラリを使います。また、(ポリゴンの) 面積や (ポリゴンや線などの) 長さのようなジオメトリの幾何学的なプロパティを計算できます。

ここでは、与えられたレイヤ内の地物を繰り返し処理し、そのジオメトリに基づいていくつかの幾何学的な計算を組み合わせた簡単な例があります。

```
# we assume that 'layer' is a polygon layer
features = layer.getFeatures()
for f in features:
  geom = f.geometry()
  print "Area:", geom.area()
  print "Perimeter:", geom.length()
```

`QgsGeometry` クラスのこれらのメソッドを使って計算するとき、面積と周長は CRS を考慮しません。より強力な面積と距離計算のために、 `QgsDistanceArea` クラスが使うことができます。投影法が切り替わったら計算は平面的に行われます。そうでないと楕円体上で計算されます。楕円体がはっきりとセットされないとき、WGS84 パラメータが計算のために使われます。

```
d = QgsDistanceArea()
d.setProjectionsEnabled(True)

print "distance in meters: ", d.measureLine(QgsPoint(10,10),QgsPoint(11,11))
```

あなたは、QGIS に含まれているアルゴリズムの多くの例を見つけて、ベクタデータを分析し、変換するためにこれらのメソッドを使用することができます。ここにはそれらのいくつかのコードへのリンクを記載します。

Additional information can be found in following sources:

- Geometry transformation: Reproject algorithm
- Distance and area using the `QgsDistanceArea` class: Distance matrix algorithm
- Multi-part to single-part algorithm

# Chapter 7

# 投影法サポート

## 7.1 空間参照系

空間参照系 (CRS) は `QgsCoordinateReferenceSystem` クラスによってカプセル化されています。このクラスのインスタンスの作成方法はいくつかあります:

- specify CRS by its ID

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

  QGIS は参照系ごとに 3 種類の ID を使います。

  - `PostgisCrsId` — IDs used within PostGIS databases.
  - `InternalCrsId` — IDs internally used in QGIS database.
  - `EpsgCrsId` — IDs assigned by the EPSG organization

  2 番目のパラメータが指定されなければ、PostGIS SRID がデフォルトで使用されます。

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

  ```
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
        PRIMEM["Greenwich", 0.0], UNIT["degree",0.0174532925199433295],'
        AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
  crs = QgsCoordinateReferenceSystem(wkt)
  ```

- create invalid CRS and then use one of the `create*()` functions to initialize it. In following example we use Proj4 string to initialize the projection

  ```
  crs = QgsCoordinateReferenceSystem()
  crs.createFromProj4("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
  ```

CRS の作成 (例:データベース内のルックアップ) が成功したかどうかをチェックするのは賢明です: `isValid()` は `True` を返さなければなりません。

Note that for initialization of spatial reference systems QGIS needs to look up appropriate values in its internal database `srs.db`. Thus in case you create an independent application you need to set paths correctly with `QgsApplication.setPrefixPath()` otherwise it will fail to find the database. If you are running the commands from QGIS python console or developing a plugin you do not care: everything is already set up for you.

Accessing spatial reference system information

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
```

```
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

## 7.2 投影法

You can do transformation between different spatial reference systems by using `QgsCoordinateTransform` class. The easiest way to use it is to create source and destination CRS and construct `QgsCoordinateTransform` instance with them. Then just repeatedly call `transform()` function to do the transformation. By default it does forward transformation, but it is capable to do also inverse transformation

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

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

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

# Chapter 8

# マップキャンバスの利用

The Map canvas widget is probably the most important widget within QGIS because it shows the map composed from overlaid map layers and allows interaction with the map and layers. The canvas shows always a part of the map defined by the current canvas extent. The interaction is done through the use of **map tools**: there are tools for panning, zooming, identifying layers, measuring, vector editing and others. Similar to other graphics programs, there is always one tool active and the user can switch between the available tools.

Map canvas is implemented as `QgsMapCanvas` class in `qgis.gui` module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please make sure to read the overview of the framework.

Whenever the map has been panned, zoomed in/out (or some other action triggers a refresh), the map is rendered again within the current extent. The layers are rendered to an image (using `QgsMapRenderer` class) and that image is then displayed in the canvas. The graphics item (in terms of the Qt graphics view framework) responsible for showing the map is `QgsMapCanvasMap` class. This class also controls refreshing of the rendered map. Besides this item which acts as a background, there may be more **map canvas items**. Typical map canvas items are rubber bands (used for measuring, vector editing etc.) or vertex markers. The canvas items are usually used to give some visual feedback for map tools, for example, when creating a new polygon, the map tool creates a rubber band canvas item that shows the current shape of the polygon. All map canvas items are subclasses of `QgsMapCanvasItem` which adds some more functionality to the basic `QGraphicsItem` objects.

要約すると、マップキャンバスアーキテクチャは3つのコンセプトからなります：

- マップキャンバス — 地図の可視化
- マップキャンバスアイテム—マップキャンバスで表示できる追加アイテム
- マップツールズ—マップキャンバスのインタラクション

## 8.1 マップキャンバスの埋め込み

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

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

This produces a standalone window with map canvas. It can be also embedded into an existing widget or window. When using .ui files and Qt Designer, place a `QWidget` on the form and promote it to a new class: set `QgsMapCanvas` as class name and set `qgis.gui` as header file. The `pyuic4` utility will take care of it. This is a very convenient way of embedding the canvas. The other possibility is to manually write the code to construct map canvas and other widgets (as children of a main window or dialog) and create a layout.

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

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

(In case you are wondering, `Qt` comes from `PyQt4.QtCore` module and `Qt.white` is one of the predefined `QColor` instances.)

Now it is time to add some map layers. We will first open a layer and add it to the map layer registry. Then we will set the canvas extent and set the list of layers for canvas

```
layer = QgsVectorLayer(path, name, provider)
if not layer.isValid():
  raise IOError, "Failed to open the layer"

# add layer to the registry
QgsMapLayerRegistry.instance().addMapLayer(layer)

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

# set the map canvas layer set
canvas.setLayerSet([QgsMapCanvasLayer(layer)])
```

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

## 8.2 マップキャンバスでのマップツールズの利用

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

```
from qgis.gui import *
from PyQt4.QtGui import QAction, QMainWindow
from PyQt4.QtCore import SIGNAL, Qt, QString

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.setLayerSet([QgsMapCanvasLayer(layer)])

    self.setCentralWidget(self.canvas)

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

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

    self.connect(actionZoomIn, SIGNAL("triggered()"), self.zoomIn)
```

```
    self.connect(actionZoomOut, SIGNAL("triggered()"), self.zoomOut)
    self.connect(actionPan, SIGNAL("triggered()"), self.pan)

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

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

    self.pan()

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

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

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

You can put the above code to a file, e.g. `mywnd.py` and try it out in Python console within QGIS. This code will put the currently selected layer into newly created canvas

```
import mywnd
w = mywnd.MyWnd(qgis.utils.iface.activeLayer())
w.show()
```

Just make sure that the `mywnd.py` file is located within Python search path (`sys.path`). If it isn't, you can simply add it: `sys.path.insert(0, '/my/path')` — otherwise the import statement will fail, not finding the module.

## 8.3 ラバーバンドと頂点マーカー

To show some additional data on top of the map in canvas, use map canvas items. It is possible to create custom canvas item classes (covered below), however there are two useful canvas item classes for convenience: `QgsRubberBand` for drawing polylines or polygons, and `QgsVertexMarker` for drawing points. They both work with map coordinates, so the shape is moved/scaled automatically when the canvas is being panned or zoomed.

To show a polyline

```
r = QgsRubberBand(canvas, False)  # False = not a polygon
points = [QgsPoint(-1, -1), QgsPoint(0, 1), QgsPoint(1, -1)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

To show a polygon

```
r = QgsRubberBand(canvas, True)  # True = a polygon
points = [[QgsPoint(-1, -1), QgsPoint(0, 1), QgsPoint(1, -1)]]
r.setToGeometry(QgsGeometry.fromPolygon(points), None)
```

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

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

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

The canvas items are bound to the canvas scene. To temporarily hide them (and show again, use the `hide()` and `show()` combo. To completely remove the item, you have to remove it from the scene of the canvas

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

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

Rubber band can be also used for drawing points, however `QgsVertexMarker` class is better suited for this (`QgsRubberBand` would only draw a rectangle around the desired point). How to use the vertex marker

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPoint(0, 0))
```

This will draw a red cross on position [0,0]. It is possible to customize the icon type, size, color and pen width

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

For temporary hiding of vertex markers and removing them from canvas, the same applies as for the rubber bands.

## 8.4 カスタムマップツールズの書き込み

You can write your custom tools, to implement a custom behaviour to actions performed by users on the canvas.

Map tools should inherit from the `QgsMapTool` class or any derived class, and selected as active tools in the canvas using the `setMapTool()` method as we have already seen.

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

```
class RectangleMapTool(QgsMapToolEmitPoint):
  def __init__(self, canvas):
      self.canvas = canvas
      QgsMapToolEmitPoint.__init__(self, self.canvas)
      self.rubberBand = QgsRubberBand(self.canvas, QGis.Polygon)
      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(QGis.Polygon)

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

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

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

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

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

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

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

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

      return QgsRectangle(self.startPoint, self.endPoint)

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

## 8.5 カスタムマップキャンバスアイテムの書き込み

**TODO:** how to create a map canvas item

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

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

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

# Chapter 9

# 地図のレンダリングと印刷

There are generally two approaches when input data should be rendered as a map: either do it quick way using `QgsMapRenderer` or produce more fine-tuned output by composing the map with `QgsComposition` class and friends.

## 9.1 単純なレンダリング

Render some layers using `QgsMapRenderer` — create destination paint device (`QImage`, `QPainter` etc.), set up layer set, extent, output size and do the rendering

```
# create image
img = QImage(QSize(800, 600), QImage.Format_ARGB32_Premultiplied)

# set image's background color
color = QColor(255, 255, 255)
img.fill(color.rgb())

# create painter
p = QPainter()
p.begin(img)
p.setRenderHint(QPainter.Antialiasing)

render = QgsMapRenderer()

# set layer set
lst = [layer.getLayerID()]  # add ID of every layer
render.setLayerSet(lst)

# set extent
rect = QgsRect(render.fullExtent())
rect.scale(1.1)
render.setExtent(rect)

# set output size
render.setOutputSize(img.size(), img.logicalDpiX())

# do the rendering
render.render(p)

p.end()

# save image
img.save("render.png","png")
```

## 9.2 Rendering layers with different CRS

If you have more than one layer and they have a different CRS, the simple example above will probably not work: to get the right values from the extent calculations you have to explicitly set the destination CRS and enable OTF reprojection as in the example below (only the renderer configuration part is reported)

```
...
# set layer set
layers = QgsMapLayerRegistry.instance().mapLayers()
lst = layers.keys()
render.setLayerSet(lst)

# Set destination CRS to match the CRS of the first layer
render.setDestinationCrs(layers.values()[0].crs())
# Enable OTF reprojection
render.setProjectionsEnabled(True)
...
```

## 9.3 マップコンポーザを使った出力

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

The composer consists of a bunch of classes. They all belong to the core library. QGIS application has a convenient GUI for placement of the elements, though it is not available in the GUI library. If you are not familiar with Qt Graphics View framework, then you are encouraged to check the documentation now, because the composer is based on it.

The central class of the composer is `QgsComposition` which is derived from `QGraphicsScene`. Let us create one

```
mapRenderer = iface.mapCanvas().mapRenderer()
c = QgsComposition(mapRenderer)
c.setPlotStyle(QgsComposition.Print)
```

Note that the composition takes an instance of `QgsMapRenderer`. In the code we expect we are running within QGIS application and thus use the map renderer from map canvas. The composition uses various parameters from the map renderer, most importantly the default set of map layers and the current extent. When using composer in a standalone application, you can create your own map renderer instance the same way as shown in the section above and pass it to the composition.

It is possible to add various elements (map, label, ...) to the composition — these elements have to be descendants of `QgsComposerItem` class. Currently supported items are:

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

  ```
  x, y = 0, 0
  w, h = c.paperWidth(), c.paperHeight()
  composerMap = QgsComposerMap(c, x ,y, w, h)
  c.addItem(composerMap)
  ```

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

  ```
  composerLabel = QgsComposerLabel(c)
  composerLabel.setText("Hello world")
  composerLabel.adjustSizeToText()
  c.addItem(composerLabel)
  ```

- legend

```
legend = QgsComposerLegend(c)
legend.model().setLayerSet(mapRenderer.layerSet())
c.addItem(legend)
```

- scale bar

```
item = QgsComposerScaleBar(c)
item.setStyle('Numeric') # optionally modify the style
item.setComposerMap(composerMap)
item.applyDefaultSize()
c.addItem(item)
```

- 矢印
- ピクチャ
- 図形
- テーブル

By default the newly created composer items have zero position (top left corner of the page) and zero size. The position and size are always measured in millimeters

```
# set label 1cm from the top and 2cm from the left of the page
composerLabel.setItemPosition(20, 10)
# set both label's position and size (width 10cm, height 3cm)
composerLabel.setItemPosition(20, 10, 100, 30)
```

A frame is drawn around each item by default. How to remove the frame

```
composerLabel.setFrame(False)
```

Besides creating the composer items by hand, QGIS has support for composer templates which are essentially compositions with all their items saved to a .qpt file (with XML syntax). Unfortunately this functionality is not yet available in the API.

Once the composition is ready (the composer items have been created and added to the composition), we can proceed to produce a raster and/or vector output.

The default output settings for composition are page size A4 and resolution 300 DPI. You can change them if necessary. The paper size is specified in millimeters

```
c.setPaperSize(width, height)
c.setPrintResolution(dpi)
```

### 9.3.1 ラスタイメージへの出力

The following code fragment shows how to render a composition to a raster image

```
dpi = c.printResolution()
dpmm = dpi / 25.4
width = int(dpmm * c.paperWidth())
height = int(dpmm * c.paperHeight())

# create output image and initialize it
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

# render the composition
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
```

```
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()

image.save("out.png", "png")
```

### 9.3.2 PDF への出力

The following code fragment renders a composition to a PDF file

```
printer = QPrinter()
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName("out.pdf")
printer.setPaperSize(QSizeF(c.paperWidth(), c.paperHeight()), QPrinter.Millimeter)
printer.setFullPage(True)
printer.setColorMode(QPrinter.Color)
printer.setResolution(c.printResolution())

pdfPainter = QPainter(printer)
paperRectMM = printer.pageRect(QPrinter.Millimeter)
paperRectPixel = printer.pageRect(QPrinter.DevicePixel)
c.render(pdfPainter, paperRectPixel, paperRectMM)
pdfPainter.end()
```

# Chapter 10

# 表現、フィルタリング及び値の算出

QGIS has some support for parsing of SQL-like expressions. Only a small subset of SQL syntax is supported. The expressions can be evaluated either as boolean predicates (returning True or False) or as functions (returning a scalar value).

3 つの基本的な種別がサポートされています:

- 数値 — 実数及び 10 進数。例. `123`, `3.14`
- 文字列 — シングルクオートで囲む必要があります: `'hello world'`
- カラム参照 — 評価する際に、参照は項目の実際の値で置き換えられます。名前はエスケープされません。

次の演算子が利用可能です:

- 算術演算子: +, –, *, /, ^
- 丸括弧: 演算を優先します: `(1 + 1) * 3`
- 単項のプラスとマイナス: `-12`, `+5`
- 数学的ファンクション: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- ジオメトリファンクション: `$area`, `$length`
- conversion functions: `to int`, `to real`, `to string`

以下の記述がサポートされています:

- 比較: =, !=, >, >=, <, <=
- パターンマッチング: `LIKE` (% と _ を使用), ~ (正規表現)
- 論理記述: `AND`, `OR`, `NOT`
- NULL 値チェック: `IS NULL`, `IS NOT NULL`

記法例:

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

スカラー表現の例:

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

## 10.1 パース表現

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> exp.hasParserError()
False
>>> exp = QgsExpression('1 + 1 = ')
>>> exp.hasParserError()
True
>>> exp.parserErrorString()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

## 10.2 評価表現

### 10.2.1 基本表現

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

### 10.2.2 地物に関わる表現

次の例は機能に対して与えられた表現を評価しています。"Column" はレイヤ内の項目名です。

```
>>> exp = QgsExpression('Column = 99')
>>> value = exp.evaluate(feature, layer.pendingFields())
>>> bool(value)
True
```

複数の地物をチェックする必要がある場合は、 `QgsExpression.prepare()` も使うことができます。`QgsExpression.prepare()` を使うと、実行の評価速度を向上できます。

```
>>> exp = QgsExpression('Column = 99')
>>> exp.prepare(layer.pendingFields())
>>> value = exp.evaluate(feature)
>>> bool(value)
True
```

### 10.2.3 エラー処理

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

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

print value
```

## 10.3 例

次の例はレイヤをフィルタリングして記法にマッチする任意の地物を返却します。

```
def where(layer, exp):
  print "Where"
  exp = QgsExpression(exp)
  if exp.hasParserError():
    raise Exception(exp.parserErrorString())
  exp.prepare(layer.pendingFields())
  for feature in layer.getFeatures():
    value = exp.evaluate(feature)
    if exp.hasEvalError():
      raise ValueError(exp.evalErrorString())
    if bool(value):
      yield feature

layer = qgis.utils.iface.activeLayer()
for f in where(layer, 'Test > 1.0'):
  print f + " Matches expression"
```

# Chapter 11

# 設定の読み込みと保存

それは、何回もユーザーがプラグインの実行される次回の日時を入力したり、それらを再度選択する必要がないように、いくつかの変数を保存するためのプラグインのために便利です。

These variables can be saved and retrieved with help of Qt and QGIS API. For each variable, you should pick a key that will be used to access the variable — for user's favourite color you could use key "favourite_color" or any other meaningful string. It is recommended to give some structure to naming of keys.

我々は、異なる種類の設定をすることができます：

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of `QSettings` class. By default, this class stores settings in system's "native" way of storing settings, that is — registry (on Windows), .plist file (on Mac OS X) or .ini file (on Unix). The QSettings documentation is comprehensive, so we will provide just a simple example

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

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

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

- **project settings** — vary between different projects and therefore they are connected with a project file. Map canvas background color or destination coordinate reference system (CRS) are examples — white background and WGS84 might be suitable for one project, while yellow background and UTM projection are better for another one. An example of usage follows

  ```
  proj = QgsProject.instance()

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

  # read values
  mytext = proj.readEntry("myplugin", "mytext", "default text")[0]
  myint = proj.readNumEntry("myplugin", "myint", 123)[0]
  ```

ご覧の通り `writeEntry()` メソッドがすべてのデータ型のために使われますが、いくつかのメソッドが後ろに設定値を読み込むために存在し、対応するものは各々のデータ型のために選ばなければなりません。

- **map layer settings** — these settings are related to a particular instance of a map layer with a project. They are *not* connected with underlying data source of a layer, so if you create two map layer instances of one shapefile, they will not share the settings. The settings are stored in project file, so if the user opens the project again, the layer-related settings will be there again. This functionality has been added in QGIS v1.4. The API is similar to QSettings — it takes and returns QVariant instances

```
# save a value
layer.setCustomProperty("mytext", "hello world")

# read the value again
mytext = layer.customProperty("mytext", "default text")
```

# Chapter 12

# ユーザとのコミュニケーション

このセクションではユーザインターフェースにおいて継続性を維持するためにユーザとのコミュニケーション時に使うべきメソッドとエレメントをいくつか示しています。

## 12.1 メッセージ表示中。QgsMessageBar クラス。

メッセージボックスの使用はユーザ体験の見地からは良いアイデアではありません。情報、警告/エラー用の小さな表示行には、たいてい QGIS メッセージバーが良い選択肢です。

QGIS インタフェースオブジェクトへの参照を利用すると、次のようなコードでメッセージバー内にメッセージを表示することができます。

```
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that", level=QgsMes
```

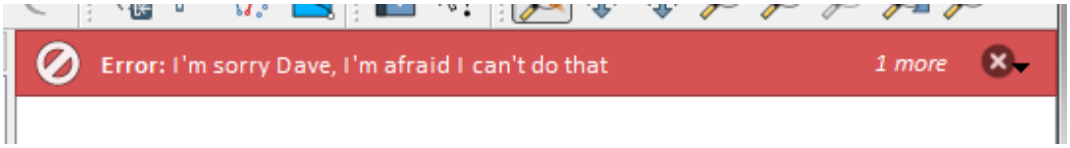


Figure 12.1: QGIS メッセージバー

表示期間を設定して時間を限定することができます。

```
iface.messageBar().pushMessage("Error", ""Ooops, the plugin is not working as it should", level=Qg
```

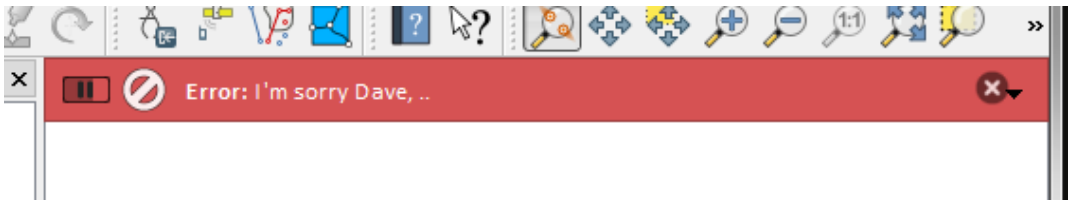


Figure 12.2: ターマー付き QGIS メッセージバー

上記の例はエラーバーを示していますが、 `level` パラメータは `QgsMessageBar.WARNING` および `QgsMessageBar.INFO` 定数をそれぞれに使って、警告メッセージやお知らせメッセージを作成するのに使うことができます。

ウィジェットは、例えば詳細情報の表示用ボタンのように、メッセージバーに追加することができます

```
def showError():
    pass
```

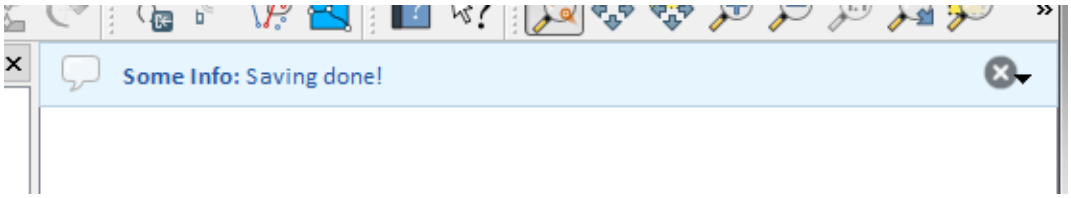


Figure 12.3: QGIS メッセージバー (お知らせ)

```
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, QgsMessageBar.WARNING)
```

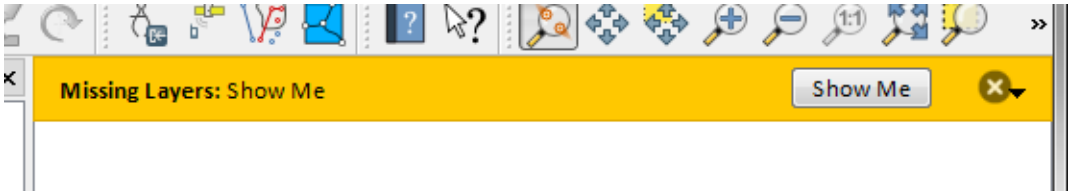


Figure 12.4: ボタン付きの QGIS メッセージバー

メッセージバーは自分のダイアログの中でも使えるため、メッセージボックスを表示する必要はありませんし、メインの QGIS ウィンドウ内に表示する意味がない時にも使えます。

```
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=QgsMessageBar.INFO)
```

## 12.2 プロセス表示中

ご覧のとおりウィジェットを受け入れるので、プログレスバーは QGIS メッセージバーに置くこともできます。コンソール内で試すことができる例はこちらです。

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.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, iface.messageBar().INFO)
```

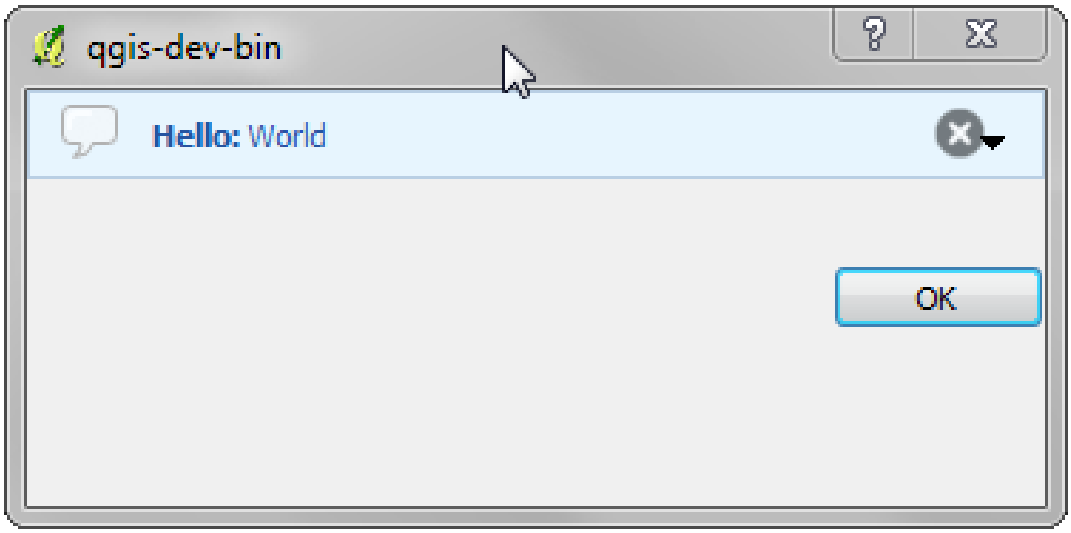


Figure 12.5: カスタムダイアログ内の QGIS メッセージバー

```
for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

次の例のように、ビルトインステータスバーを使って進捗をレポートすることもできます

```
count = layers.featureCount()
for i, feature in enumerate(features):
    #do something time-consuming here
    ...
    percent = i / float(count) * 100
    iface.mainWindow().statusBar().showMessage("Processed {} %".format(int(percent)))
iface.mainWindow().statusBar().clearMessage()
```

## 12.3 ロギング

QGIS ロギングシステムを使うとコードの実行に関して保存したい情報のログを全て採ることができます。

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

# Chapter 13

# Python プラグインの開発

Python プログラミング言語でプラグインを作成することが可能です。C++で書かれた古典的なプラグインと比較して、これらは Python 言語の動的な性質により、記述や理解、維持、配布が簡単です。

Python plugins are listed together with C++ plugins in QGIS plugin manager. They are searched for in these paths:

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

Home directory (denoted by above ~) on Windows is usually something like `C:\Documents and Settings\(user)` (on Windows XP or earlier) or `C:\Users\(user)`. Since QGIS is using Python 2.7, subdirectories of these paths have to contain an __init__.py file to be considered Python packages that can be imported as plugins.

---

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

---

ステップ：

1. **アイデア**: 新しい QGIS プラグインでやりたいことのアイデアを持ちます。なぜそれを行うのですか? どのような問題を解決しますか? その問題のための別のプラグインは既にありますか?
2. *Create files*: Create the files described next. A starting point (`__init__.py`). Fill in the **プラグインメタデータ** (`metadata.txt`) A main python plugin body (`mainplugin.py`). A form in QT-Designer (`form.ui`), with its `resources.qrc`.
3. **コードを書く**: `mainplugin.py` 内にコードを記述する
4. **テスト**: QGIS を閉じて再度開き、あなたのプラグインをインポートします。すべてが OK かチェックして下さい。
5. *Publish*: Publish your plugin in QGIS repository or make your own repository as an “arsenal” of personal “GIS weapons”.

## 13.1 プラグインを書く

Since the introduction of Python plugins in QGIS, a number of plugins have appeared - on Plugin Repositories wiki page you can find some of them, you can use their source to learn more about programming with PyQGIS or find out whether you are not duplicating development effort. The QGIS team also maintains an **公式の** *python* **プラグインリポジトリ**. Ready to create a plugin but no idea what to do? Python Plugin Ideas wiki page lists wishes from the community!

### 13.1.1 プラグインファイル

Here's the directory structure of our example plugin

```
PYTHON_PLUGINS_PATH/
  MyPlugin/
    __init__.py     --> *required*
    mainPlugin.py   --> *required*
    metadata.txt    --> *required*
    resources.qrc   --> *likely useful*
    resources.py    --> *compiled version, likely useful*
    form.ui         --> *likely useful*
    form.py         --> *compiled version, likely useful*
```

ファイルが意味すること:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.py` = プラグインの主なワーキングコード。このプラグインの動作に関するすべての情報と主要なコードを含みます。
- `resources.qrc` = The .xml document created by Qt Designer. Contains relative paths to resources of the forms.
- `resources.py` = 上記の.qrc ファイルが Python に変換されたもの。
- `form.ui` = The GUI created by Qt Designer.
- `form.py` = 上記の form.ui が Python に変換されたもの。
- `metadata.txt` = Required for QGIS >= 1.8.0. Containts general info, version, name and some other metadata used by plugins website and plugin infrastructure. Since QGIS 2.0 the metadata from `__init__.py` are not accepted anymore and the `metadata.txt` is required.

ここ では典型的な QGIS の Python プラグインの基本的なファイル (スケルトン) をオンラインで自動的に作成することができます。

また、 Plugin Builder と呼ばれる QGIS プラグインがあります。QGIS からプラグインテンプレートを作成しますがインターネット接続を必要としません。2.0 に互換性のあるソースを生成しますので推奨される選択肢です。

**警告:** If you plan to upload the plugin to the **公式の *python* プラグインリポジトリ** you must check that your plugin follows some additional rules, required for plugin **検証**

## 13.2 プラグインの内容

上述のファイル構造の中のそれぞれのファイルに何を追加するべきかについての情報および例を示します。

### 13.2.1 プラグインメタデータ

まず、プラグインマネージャーは名前や説明などプラグインに関する基本的な情報を取得する必要があります。 `metadata.txt` ファイルはこの情報を記載するのに適切な場所です。

**重要:** 全てのメタデータは UTF-8 のエンコーディングでなければいけません。

| メタデータ名 | 必須 | 注意 |
|---|---|---|
| name | True | プラグインの名前を含んでいる短い文字列 |
| qgisMinimumVersion | True | QGIS の最小バージョンのドット付き表記 |
| qgisMaximumVersion | False | QGIS の最大バージョンのドット付き表記 |
| description | True | プラグインを説明する短いテキスト。HTML は使用できません。 |
| about | False | プラグインを詳細に説明するより長いテキスト。HTML は使用できません。 |
| version | True | バージョンのドット付き表記の短い文字列 |
| author | True | 作者名 |
| email | True | email of the author, will *not* be shown on the web site |
| changelog | False | 文字列。複数行でもよいですが HTML は使用できません。 |
| experimental | False | ブール型のフラグ。 *True* または *False* |
| deprecated | False | ブール型のフラグ。 *True* または *False*. アップロードされたバージョンだけではなくプラグイン全体に適用されます。 |
| tags | False | comma separated list, spaces are allowed inside individual tags |
| homepage | False | プラグインのホームページを指す有効な URL |
| repository | False | ソースコードリポジトリの有効な URL |
| tracker | False | チケットとバグ報告のための有効な URL |
| icon | False | a file name or a relative path (relative to the base folder of the plugin's compressed package) |
| category | False | *Raster*, *Vector*, *Database*, *Web* のいずれか |

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

A corresponding "category" metadata entry exists to specify that, so the plugin can be classified accordingly. This metadata entry is used as tip for users and tells them where (in which menu) the plugin can be found. Allowed values for "category" are: Vector, Raster, Database or Web. For example, if your plugin will be available from *Raster* menu, add this to `metadata.txt`

```
category=Raster
```

---

**ノート:** *qgisMaximumVersion* が空の場合、 **公式の** *python* **プラグインリポジトリ** にアップロードされた時にメジャーバージョン + *.99* に自動的に設定されます。

---

An example for this metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
; 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=http://www.itopen.it
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
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=2.0
```

### 13.2.2 __init__.py

This file is required by Python's import system. Also, QGIS requires that this file contains a `classFactory()` function, which is called when the plugin gets loaded to QGIS. It receives reference to instance of `QgisInterface` and must return instance of your plugin's class from the `mainplugin.py` — in our case it's called `TestPlugin` (see below). This is how `__init__.py` should look like

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

## any other initialisation needed
```

### 13.2.3 mainPlugin.py

This is where the magic happens and this is how magic looks like: (e.g. `mainPlugin.py`)

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

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

```
    QObject.connect(self.action, SIGNAL("triggered()"), 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
    QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), 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
    QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTe

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

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

The only plugin functions that must exist in the main plugin source file (e.g. `mainPlugin.py`) are:

- `__init__` –> which gives access to QGIS interface
- `initGui()` –> called when the plugin is loaded
- `unload()` –> called when the plugin is unloaded

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

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

それらはすべて `addPluginToMenu()` メソッドと同じ構文です。

プラグインエントリの編成の一貫性を保つために、これらの定義済みのメソッドのいずれかでプラグインメニューを追加することが推奨されます。ただし、次の例に示すようにメニューバーに直接カスタムメニューグループを追加することができます:

```
def initGui(self):
    self.menu = QMenu(self.iface.mainWindow())
    self.menu.setObjectName("testMenu")
    self.menu.setTitle("MyMenu")

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWin
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)
    self.menu.addAction(self.action)

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

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

Don't forget to set `QAction` and `QMenu` `objectName` to a name specific to your plugin so that it can be customized.

### 13.2.4 リソースファイル

You can see that in `initGui()` we've used an icon from the resource file (called `resources.qrc` in our case)

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

It is good to use a prefix that will not collide with other plugins or any parts of QGIS, otherwise you might get resources you did not want. Now you just need to generate a Python file that will contain the resources. It's done with **pyrcc4** command

```
pyrcc4 -o resources.py resources.qrc
```

And that's all... nothing complicated :)

If you've done everything correctly you should be able to find and load your plugin in the plugin manager and see a message in console when toolbar icon or appropriate menu item is selected.

本物のプラグインに取り組んでいる時は別の (作業) ディレクトリでプラグインを書いて、UI とリソースファイルを生成してプラグインを QGIS にインストールする makefile を作成するのが賢明です。

## 13.3 ドキュメント

プラグインのドキュメントは HTML ヘルプファイルとして記述できます。`qgis.utils` モジュールは他の QGIS のヘルプと同じ方法でヘルプファイルブラウザを開く `showPluginHelp()` 関数を提供しています。

The `showPluginHelp`()` function looks for help files in the same directory as the calling module. It will look for, in turn, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` and `index.html`, displaying whichever it finds first. Here `ll_cc` is the QGIS locale. This allows multiple translations of the documentation to be included with the plugin.

`showPluginHelp()` 関数は引数をとることができます。packageName 引数はヘルプが表示されるプラグインを識別します。filename 引数は検索しているファイル名の"index" を置き換えます。そして section 引数はブラウザが表示位置を合わせるドキュメント内の HTML アンカータグの名前です。

# Chapter 14

# 書き込みのIDE設定とデバッグプラグイン

Although each programmer has his preferred IDE/Text editor, here are some recommendations for setting up popular IDE's for writing and debugging QGIS Python plugins.

## 14.1 Windows 上で IDE を設定するメモ

On Linux there is no additional configuration needed to develop plug-ins. But on Windows you need to make sure you that you have the same environment settings and use the same libraries and interpreter as QGIS. The fastest way to do this, is to modify the startup batch file of QGIS.

If you used the OSGeo4W Installer, you can find this under the bin folder of your OSGoeW install. Look for something like `C:\OSGeo4W\bin\qgis-unstable.bat`.

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

- Make a copy of qgis-unstable.bat and rename it pyscripter.bat.
- Open it in an editor. And remove the last line, the one that starts QGIS.
- Add a line that points to the your Pyscripter executable and add the commandline argument that sets the version of Python to be used (2.7 in the case of QGIS 2.0)
- Also add the argument that points to the folder where Pyscripter can find the Python dll used by QGIS, you can find this under the bin folder of your OSGeoW install

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

Now when you double click this batch file it will start Pyscripter, with the correct path.

More popular that Pyscripter, Eclipse is a common choice among developers. In the following sections, we will be explaining how to configure it for developing and testing plugins. To prepare your environment for using Eclipse in Windows, you should also create a batch file and use it to start Eclipse.

To create that batch file, follow these steps.

- Locate the folder where file:*qgis_core.dll* resides in. Normally this is `C:OSGeo4Wappsqgisbin`, but if you compiled your own QGIS application this is in your build folder in `output/bin/RelWithDebInfo`
- :file:'eclipse.exe'を実行可能にします。

- QGIS プラグインを開発するとき、以下のスクリプトを作成して、 eclipse のスタート時にこれを使ってください。

```
call "C:\OSGeo4W\bin\o4w_env.bat"
set PATH=%PATH%;C:\path\to\your\qgis_core.dll\parent\folder
C:\path\to\your\eclipse.exe
```

## 14.2 Eclipse と PyDev を利用したデバッグ

### 14.2.1 インストール

Eclipse を使用するため、あなたが以下をインストールしたことを確認してください

- Eclipse
- Aptana Eclipse Plugin or PyDev
- QGIS 2.0

### 14.2.2 QGIS の準備

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

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

### 14.2.3 Eclipse のセットアップ

In Eclipse, create a new project. You can select *General Project* and link your real sources later on, so it does not really matter where you place this project.

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

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

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

### 14.2.4 デバッガの設定

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

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

Eclipse is now waiting for a connection from QGIS to its debug server and when QGIS connects to the debug server it will allow it to control the python scripts. That's exactly what we installed the *Remote Debug* plugin for. So start QGIS in case you did not already and click the bug symbol .

Now you can set a breakpoint and as soon as the code hits it, execution will stop and you can inspect the current state of your plugin. (The breakpoint is the green dot in the image below, set one by double clicking in the white space left to the line you want the breakpoint to be set)

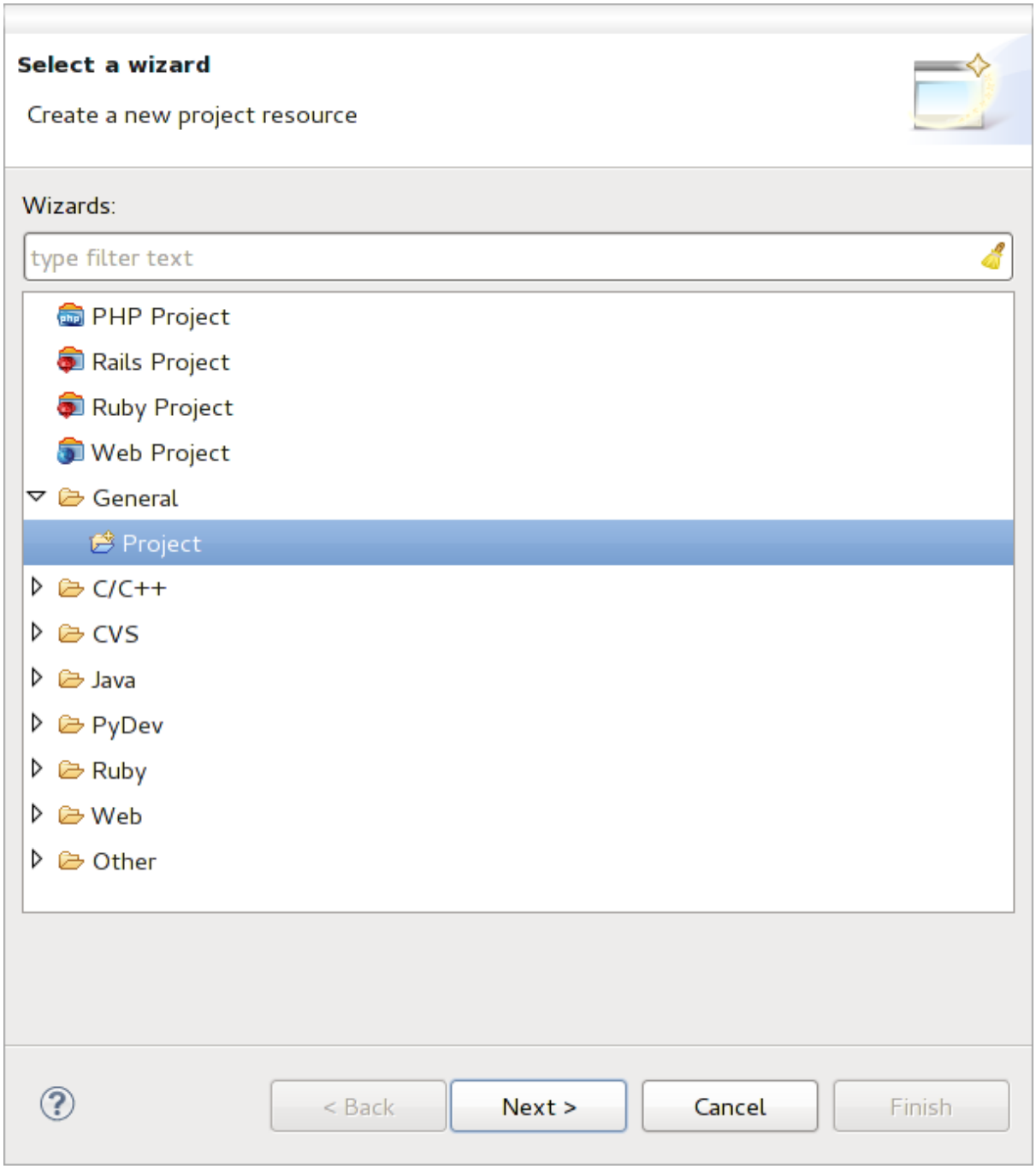


Figure 14.1: Eclipse プロジェクト

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

Figure 14.2: ブレークポイント

A very interesting thing you can make use of now is the debug console. Make sure that the execution is currently stopped at a break point, before you proceed.

Open the Console view (*Window* → *Show view*). It will show the *Debug Server* console which is not very interesting. But there is a button **[Open Console]** which lets you change to a more interesting PyDev Debug Console. Click the arrow next to the **[Open Console]** button and choose *PyDev Console*. A window opens up to ask you which console you want to start. Choose *PyDev Debug Console*. In case its greyed out and tells you to Start the debugger and select the valid frame, make sure that you've got the remote debugger attached and are currently on a breakpoint.

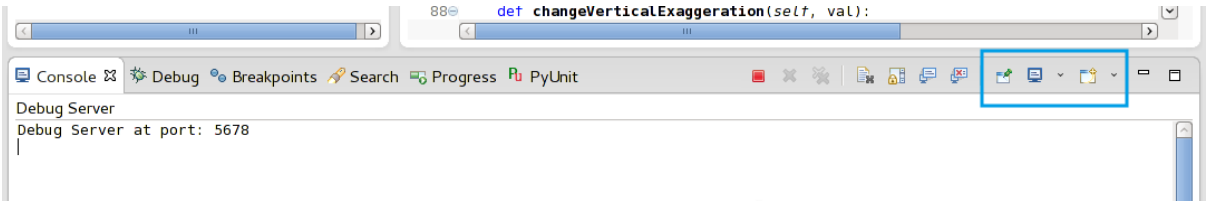


Figure 14.3: PyDev デバッグコンソール

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

A little bit annoying is, that every time you enter a command, the console switches back to the Debug Server. To stop this behavior, you can click the *Pin Console* button when on the Debug Server page and it should remember this decision at least for the current debug session.

### 14.2.5 eclipse での API の理解

A very handy feature is to have Eclipse actually know about the QGIS API. This enables it to check your code for typos. But not only this, it also enables Eclipse to help you with autocompletion from the imports to API calls.

To do this, Eclipse parses the QGIS library files and gets all the information out there. The only thing you have to do is to tell Eclipse where to find the libraries.

Click *Window* → *Preferences* → *PyDev* → *Interpreter* → *Python*.

You will see your configured python interpreter in the upper part of the window (at the moment python2.7 for QGIS) and some tabs in the lower part. The interesting tabs for us are *Libraries* and *Forced Builtins*.

First open the Libraries tab. Add a New Folder and choose the python folder of your QGIS installation. If you do not know where this folder is (it's not the plugins folder) open QGIS, start a python console and simply enter `qgis` and press Enter. It will show you which QGIS module it uses and its path. Strip the trailing `/qgis/__init__.pyc` from this path and you've got the path you are looking for.

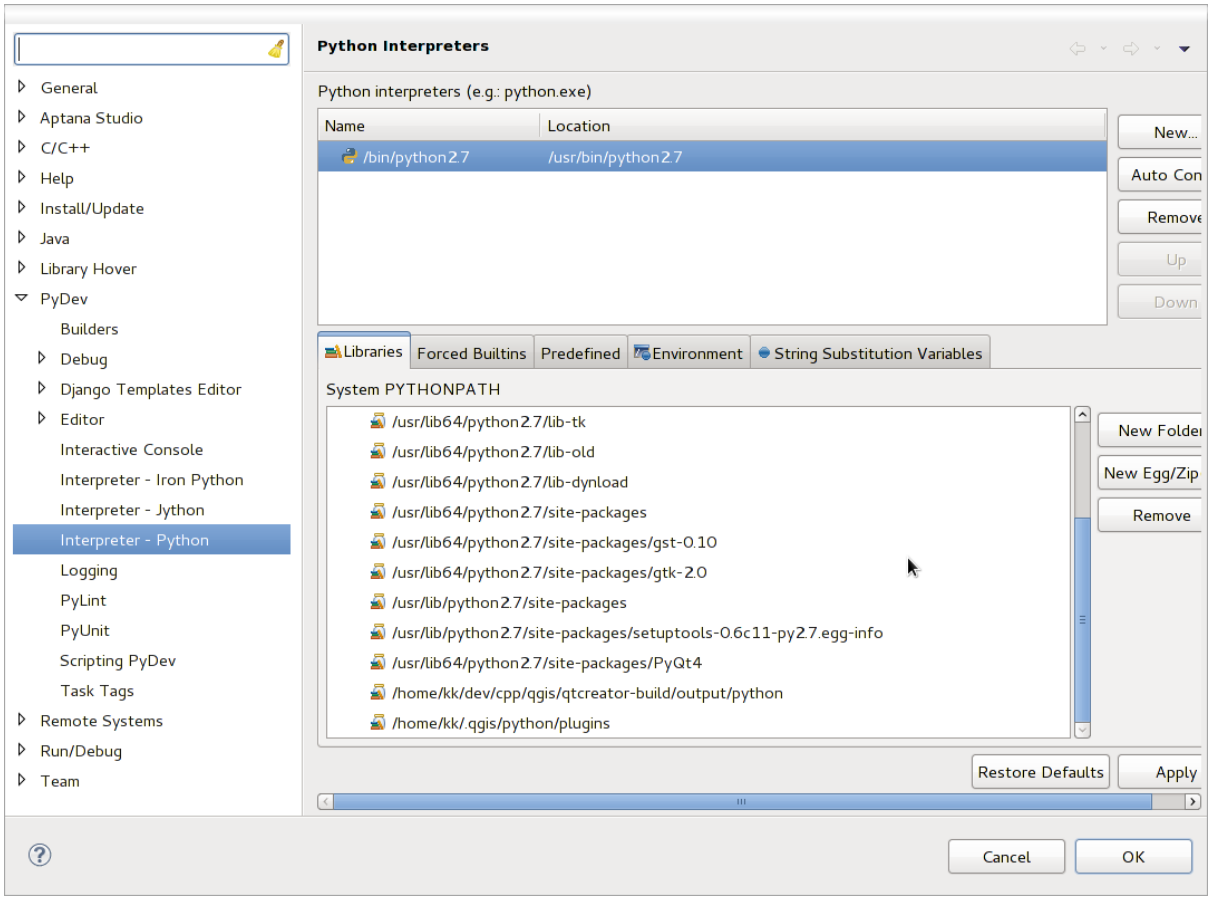


Figure 14.4: PyDev デバッグコンソール

You should also add your plugins folder here (on Linux it is `~/.qgis/python/plugins`).

Next jump to the *Forced Builtins* tab, click on *New...* and enter `qgis`. This will make Eclipse parse the QGIS API. You probably also want eclipse to know about the PyQt4 API. Therefore also add PyQt4 as forced builtin. That should probably already be present in your libraries tab.

*OK*をクリックし、完了します。

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

For another possible setting of Eclipse to work with QGIS Python plugins, check this link

## 14.3 Debugging using PDB

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

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

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

それから、コマンドラインから QGIS を実行します。

On Linux do:

**$ ./Qgis**

On Mac OS X do:

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

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

**TODO:** テスト情報の追加

# Chapter 15

# プラグインレイヤの利用

マップレイヤをレンダーするためにプラグインを使うなら、QgsPluginLayer に基づいたレイヤタイプを記述することが、最良な実装方法かもしれません。

**TODO:** QgsPluginLayer のよい利用ケースにおいて正しさと精巧さをチェックしましょう。

## 15.1 QgsPluginLayer のサブクラス化

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

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

プロジェクトファイルに固有の情報を読み書きするための方法も追加することができます

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

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

そのようなレイヤを含むプロジェクトをロードすると、 factory クラスが必要となります

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

また、レイヤーのプロパティでカスタム情報を表示するためのコードを追加することもできます

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

# Chapter 16

# QGISの旧バージョンとの互換性

## 16.1 プラグインメニュー

If you place your plugin menu entries into one of the new menus (*Raster*, *Vector*, *Database* or *Web*), you should modify the code of the `initGui()` and `unload()` functions. Since these new menus are available only in QGIS 2.0 and greater, the first step is to check that the running QGIS version has all the necessary functions. If the new menus are available, we will place our plugin under this menu, otherwise we will use the old *Plugins* menu. Here is an example for *Raster* menu

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

  # check if Raster menu available
  if hasattr(self.iface, "addPluginToRasterMenu"):
    # Raster menu and toolbar available
    self.iface.addRasterToolBarIcon(self.action)
    self.iface.addPluginToRasterMenu("&Test plugins", self.action)
  else:
    # there is no Raster menu, place plugin under Plugins menu as usual
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

  # connect to signal renderComplete which is emitted when canvas rendering is done
  QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def unload(self):
  # check if Raster menu available and remove our buttons from appropriate
  # menu and toolbar
  if hasattr(self.iface, "addPluginToRasterMenu"):
    self.iface.removePluginRasterMenu("&Test plugins", self.action)
    self.iface.removeRasterToolBarIcon(self.action)
  else:
    self.iface.removePluginMenu("&Test plugins", self.action)
    self.iface.removeToolBarIcon(self.action)

  # disconnect from signal of the canvas
  QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest
```

# Chapter 17

# あなたのプラグインのリリース

Once your plugin is ready and you think the plugin could be helpful for some people, do not hesitate to upload it to **公式の** *python* **プラグインリポジトリ**. On that page you can find also packaging guidelines about how to prepare the plugin to work well with the plugin installer. Or in case you would like to set up your own plugin repository, create a simple XML file that will list the plugins and their metadata, for examples see other plugin repositories.

Please take special care to the following suggestions:

## 17.1 Metadata and names

- avoid using a name too similar to existing plugins
- if your plugin has a similar functionality to an existing plugin, please explain the differences in the About field, so the user will know which one to use without the need to install and test it
- avoid repeating "plugin" in the name of the plugin itself
- use the description field in metadata for a 1 line description, the About field for more detailed instructions
- include a code repository, a bug tracker, and a home page; this will greatly enhance the possibility of collaboration, and can be done very easily with one of the available web infrastructures (GitHub, GitLab, Bitbucket, etc.)
- choose tags with care: avoid the uninformative ones (e.g. vector) and prefer the ones already used by others (see the plugin website)
- add a proper icon, do not leave the default one; see QGIS interface for a suggestion of the style to be used

## 17.2 Code and help

- do not include generated file (ui_*.py, resources_rc.py, generated help files⋯) and useless stuff (e.g. .gitignore) in repository
- add the plugin to the appropriate menu (Vector, Raster, Web, Database)
- when appropriate (plugins performing analyses), consider adding the plugin as a subplugin of Processing framework: this will allow users to run it in batch, to integrate it in more complex workflows, and will free you from the burden of designing an interface
- include at least minimal documentation and, if useful for testing and understanding, sample data.

## 17.3 公式の python プラグインリポジトリ

*公式の*python プラグインリポジトリは http://plugins.qgis.org/ で見つけることができます。

公式のリポジトリを使用するため、あなたは OSGEO web portal から OSGEO ID を入手しないといけません。

あなたがプラグインをアップロードしたら、それはスタッフによって承認され、あなたに通知されます。

**TODO:** Insert a link to the governance document

### 17.3.1 許可

これらのルールは、公式のプラグインリポジトリに実装されています：

- すべての登録ユーザは、新しいプラグインを追加することができます
- *スタッフ*は全てのプラグインバージョンの承認と非承認を行うことができます。
- users which have the special permission *plugins.can_approve* get the versions they upload automatically approved
- users which have the special permission *plugins.can_approve* can approve versions uploaded by others as long as they are in the list of the plugin *owners*
- 特定のプラグインは *staff*ユーザまたはプラグイン *owners*によって削除または編集できます。
- ユーザが 'plugins.can_approve'なしで新しいバージョンをアップロードした場合、プラグインのバージョンは自動的に非承認になります。

### 17.3.2 運用の信託

Staff members can grant *trust* to selected plugin creators setting *plugins.can_approve* permission through the front-end application.

The plugin details view offers direct links to grant trust to the plugin creator or the plugin *owners*.

### 17.3.3 検証

Plugin's metadata are automatically imported and validated from the compressed package when the plugin is uploaded.

Here are some validation rules that you should aware of when you want to upload a plugin on the official repository:

1. the name of the main folder containing your plugin must contain only ASCII characters (A-Z and a-z), digits and the characters underscore (_) and minus (-), also it cannot start with a digit
2. `metadata.txt` が要求されます。
3. all required metadata listed in *metadata table* must be present
4. the *version* metadata field must be unique

### 17.3.4 プラグイン構造

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

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

# Chapter 18

# コードスニペット

このセクションではプラグインの開発を容易にするコードスニペットを特集します。

## 18.1 キーボードショートカットによるメソッド呼び出し方法

プラグイン内での:func:‘initGui()‘への追加

```
self.keyAction = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.keyAction, "F7") # action1 triggered by F7 key
self.iface.addPluginToMenu("&Test plugins", self.keyAction)
QObject.connect(self.keyAction, SIGNAL("triggered()"),self.keyActionF7)
```

unload() の追加のため

```
self.iface.unregisterMainWindowAction(self.keyAction)
```

F7 キー押下時に呼び出されるメソッド

```
def keyActionF7(self):
  QMessageBox.information(self.iface.mainWindow(),"Ok", "You pressed F7")
```

## 18.2 レイヤの切り替え方法

QGIS 2.4 から凡例内のレイヤツリーへの直接アクセスを可能にする新しいレイヤツリー API があります。
アクティブレイヤの表示の切り替え方の例がこちらです。

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

## 18.3 選択した機能の属性テーブルへのアクセス方法

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
    nF = layer.selectedFeatureCount()
    if (nF > 0):
      layer.startEditing()
    ob = layer.selectedFeaturesIds()
```

```
    b = QVariant(value)
    if (nF > 1):
      for i in ob:
      layer.changeAttributeValue(int(i), 1, b) # 1 being the second column
    else:
      layer.changeAttributeValue(int(ob[0]), 1, b) # 1 being the second column
    layer.commitChanges()
    else:
      QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select at least one feature
  else:
    QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select a layer")
```

このメソッドにはパラメータがひとつ (選択された機能の属性項目用の新しい値) 必要で、右記より呼び出すことができます

```
self.changeValue(50)
```

# Chapter 19

# ネットワーク分析ライブラリ

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

- 地理データから数学的なグラフ（ポリラインベクタレイヤー）を作成します。
- implements basic methods from graph theory (currently only Dijkstra's algorithm)

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

## 19.1 一般情報

Briefly, a typical use case can be described as:

1. 地理データから地理学的なグラフ（たいていはポリラインベクタレイヤー）を作成します。
2. グラフ分析の実行
3. 分析結果の利用（例えば、これらの可視化）

## 19.2 Building a graph

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

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

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

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

Converting from a vector layer to the graph is done using the Builder programming pattern. A graph is constructed using a so-called Director. There is only one Director for now: QgsLineVectorLayerDirector. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create the graph. Currently, as in the case with the director, only one builder exists: QgsGraphBuilder, that creates QgsGraph objects. You may want to implement your own builders that will build a graphs compatible with such libraries as BGL or NetworkX.

To calculate edge properties the programming pattern strategy is used. For now only QgsDistanceArcProperter strategy is available, that takes into account the length of the route. You can implement your own strategy that

will use all necessary parameters. For example, RoadGraph plugin uses a strategy that computes travel time using edge length and speed value from attributes.

It's time to dive into the process.

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

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

Then some examples for creating a director

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

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

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

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

And here is full list of what these parameters mean:

- `vl` — vector layer used to build the graph
- `directionFieldId` — index of the attribute table field, where information about roads direction is stored. If `-1`, then don't use this info at all. An integer.
- `directDirectionValue` — field value for roads with direct direction (moving from first line point to last one). A string.
- `reverseDirectionValue` — field value for roads with reverse direction (moving from last line point to first one). A string.
- `bothDirectionValue` — field value for bidirectional roads (for such roads we can move from first point to last and from last to first). A string.
- `defaultDirection` — default road direction. This value will be used for those roads where field `directionFieldId` is not set or has some value different from any of the three values specified above. An integer. `1` indicates direct direction, `2` indicates reverse direction, and `3` indicates both directions.

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

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy

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

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

- crs —使用する空間参照系。必須の引数です。
- otfEnabled — "オンザフライ" 再投影を使うかどうか。デフォルトでは定数:*True* (OTF 使用).
- トポロジ許容値—トポロジ的な許容値です。デフォルト値は 0 です。

- 楕円体 ID — 使用する楕円体です。デフォルトは “WGS84” です。

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

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

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

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

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

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

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

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

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

## 19.3 グラフ分析

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

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

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

- only one vertex has no incoming edges — the root of the tree
- all other vertexes have only one incoming edge
- if vertex B is reachable from vertex A, then the path from A to B is the single available path and it is optimal (shortest) on this graph

To get the shortest path tree use the methods `shortestTree()` and `dijkstra()` of QgsGraphAnalyzer class. It is recommended to use method `dijkstra()` because it works faster and uses memory more efficiently.

The `shortestTree()` method is useful when you want to walk around the shortest path tree. It always creates a new graph object (QgsGraph) and accepts three variables:

- source — 入力グラフ
- startVertexIdx — ツリー上のポイントのインデックス（ツリーのルート）
- criterionNum — 使用するエッジプロパティの数（0 から始まる）。

```
tree = QgsGraphAnalyzer.shortestTree(graph, startId, 0)
```

The `dijkstra()` method has the same arguments, but returns two arrays. In the first array element i contains index of the incoming edge or -1 if there are no incoming edges. In the second array element i contains distance from the root of the tree to vertex i or DOUBLE_MAX if vertex i is unreachable from the root.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, startId, 0)
```

Here is some very simple code to display the shortest path tree using the graph created with the `shortestTree()` method (select linestring layer in TOC and replace coordinates with your own). **Warning**: use this code only as an example, it creates a lots of QgsRubberBand objects and may be slow on large data-sets.

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

Same thing but using `dijkstra()` method

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
```

```
  rb.addPoint (graph.vertex(graph.arc(edgeId).inVertex()).point())
  rb.addPoint (graph.vertex(graph.arc(edgeId).outVertex()).point())
```

### 19.3.1 Finding shortest paths

To find the optimal path between two points the following approach is used. Both points (start A and end B) are "tied" to the graph when it is built. Then using the methods `shortestTree()` or `dijkstra()` we build the shortest path tree with root in the start point A. In the same tree we also find the end point B and start to walk through the tree from point B to point A. The whole algorithm can be written as

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

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

Here is the sample code for QGIS Python Console (you will need to select linestring layer in TOC and replace coordinates in the code with yours) that uses method `shortestTree()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

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

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

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

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

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

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

if idStop == -1:
  print "Path not found"
else:
  p = []
  while (idStart != idStop):
    l = tree.vertex(idStop).inArc()
    if len(l) == 0:
      break
```

```
    e = tree.arc(l[0])
    p.insert(0, tree.vertex(e.inVertex()).point())
    idStop = e.outVertex()

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

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

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

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

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

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

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

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

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

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

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

  p.append(tStart)

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

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

### 19.3.2 Areas of availability

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

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

To find the areas of availability we can use method `dijkstra()` of the `QgsGraphAnalyzer` class. It is enough to compare the elements of the cost array with a predefined value. If cost[i] is less than or equal to a predefined value, then vertex i is inside the area of availability, otherwise it is outside.

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

Here is an example

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

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

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

rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
rb.setColor(Qt.green)
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() + delta))

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

idStart = graph.findVertex(tStart)

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

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

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

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